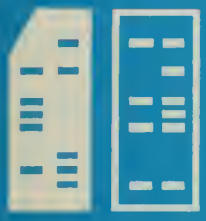
Report No. 422

AN ALGORITHM FOR EVALUATING ARRAY EXPRESSIONS IN OL/2

by

John Lockhart Latch

January 1971



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 422

AN ALGORITHM FOR EVALUATING ARRAY EXPRESSIONS IN OL/2

by

John Lockhart Latch

January 1971

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the National Science Foundation under Grant No. US NSF-GJ-328 and was submitted in partial fulfillment for the degree of Master of Science in Computer Science, January 1971.

ABSTRACT

This thesis is concerned with the evaluation of expressions in the OL/2 array language. OL/2 is a generalized array language based upon natural operations involving vectors, matrices, higher dimensional arrays, and dynamic partitioning of arrays.

The technique used for the evaluation of array expressions is divided into three phases: the parsing of source statements into an expression tree, the determination and reduction of temporary storage for intermediate results, and the generation of code to compute the expression. This technique has been implemented using the TACOS compiler-compiler and PL/1 on the IBM SYSTEM/360.

ACKNOWLEDGEMENT

I wish to thank my advisor, Professor J. R. Phillips for suggesting this thesis topic and for his guidance throughout its preparation. Additional thanks go to the other members of the OL/2 implementation team; Cory Adams, Dale Jurich, Bob Bloemer, and Barry Finkel for their cooperation and suggestions.

Furthermore, I am indebted to Mrs. Diana Mercer, who did such a fine job of typing this report, and most expecially to my wife, Fran, for her patience, understanding and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Few high level languages have the ability to handle array expressions, especially in the case of matrix algebra. Two major problems in implementing an array language involve an efficient method of calculating array operations and the minimization of temporary storage required during that calculation. The evaluation technique presented in this paper has been implemented as part of the OL/2 language and operates not only on the usual data types for array languages but also on a wider class of array data types which reflect the needs of an array language.

Several previous works in the area of array expression evaluation are of interest. Galler and Perlis [1], in 1962, described a method for calculating temporary storage requirements for a class of unparenthesized matrix expressions and proved that the method minimized the temporary storage. Reinfeld [2] discussed a method of evaluating vector expressions elementwise, with the object of reducing stemporary storage. Wagner [3] considered in detail techniques for producing algorithms for the evaluation of fully parenthesized matrix expressions involving only operands which were square matrices. He considers an algorithm which requires the fewest arrays for holding the matrices, while not requiring more execution time than "standard" matrix expression evaluation.

The most general of the previous works was described by Galler and Perlis in their recent book [4]. Their method is to extend the ALGOL language with new "definitions" which are placed in a context table along with the ALGOL construct that will replace it. The definitions are placed in the context table in a way that the precedence is determined when the table is searched from top to bottom. The source language is parsed into a tree and then matched with the syntax table. When a match occurs, the tree is changed to the desired construct. After completing translation through the context table, an expression will be represented by a tree which can be coded in ALGOL. This method seems applicable mainly to rectangular arrays. The authors do not state whether this system has been implemented, nor do they discuss the temporary storage requirements.

The array evaluation technique presented here operates on a more generalized set of data types than any of these prior works. The OL/2 language allows the declaration of vectors, various matrices, and various higher dimensional arrays. In particular, the two dimensional matrices may be defined to have a specific geometric type such as lower triangular, upper triangular, tridiagonal, diagonal, and others where only the non zero elements are stored. The operators that are allowed in OL/2 include addition, subtraction, multiplication, division, exponentiation by a scalar, as well as norm, and inner product of vectors. To efficiently perform such

operations involving different types of operands a philosophy was adopted to develop some general assembly language routines which could be optimized for array operations.

We consider the evaluation of array expressions in two parts: first the parsing of the expression into a binary tree, and second the generation of the code to compute the expression. OL/2 expressions are parsed into a binary tree by using the operator precedence relations defined in section 3. Each node of the tree contains sufficient information for code generation, including the type of node (operator or variable), the type of expression (matrix, vector or scalar), and number of dimensions.

The second section determines temporary storage requirements and generates the code to calculate the expression. The aim is to reduce temporary storage as much as possible without performing more operations than are required by "standard" matrix evaluation, and without degrading the performance of the assembly language calculating routines. The amount of temporary storage is determined as the algorithm moves systematically down the expression tree. Whenever possible the calculation is organized so that row partitions or column partitions of array variables are used at each stage of the calculation. The partition is controlled by a loop. This method reduces the temporary storage to a reasonable, if not optimal level.

The code generated consists of calls to intermediate routines, which in turn call the assembly language calculating routines. The code is produced when the terminal nodes are reached. It is also at this stage that the algorithm determines whether to partition the operands.

2. THE OL/2 LANGUAGE

The Ol/2 language allows a variety of data types. The usual matrix operations are part of the language and include vector inner product and norm. Other interesting operations such as dynamic partitioning of arrays are also provided [5]. For a description of the use of these features we refer to [6].

2.1 OL/2 Data Types

An OL/2 variable may be declared as a vector, a two dimensional matrix, or a higher dimensional array. The bounds for each dimension can be specified as constants or scalar expressions in bound pairs, the upper bound only or as the order of a matrix. One of the important features of OL/2 is to allow two dimensional arrays to be defined as a specific geometric type. The basic geometric types are lower triangular, strictly lower triangular, upper triangular, strictly upper triangular, diagonal, tridiagonal, and rectangular. In each case, only the non-zero elements are stored which reduces the total storage significantly. The user may simply use the name of an array in any OL/2 statement insuring only the compatability of dimensions; OL/2 will operate properly on any set of geometric types.

An OL/2 variable may also be defined as sequence of arrays molulo some number n, and storage is allocated for n arrays. This feature is particularly useful in iterative algorithms [6].

Another important feature of OL/2 is the dynamic partitioning. For example, Figure 1 shows a rectangular matrix A partitioned after rows K and K-1 and after columns J and J-1.

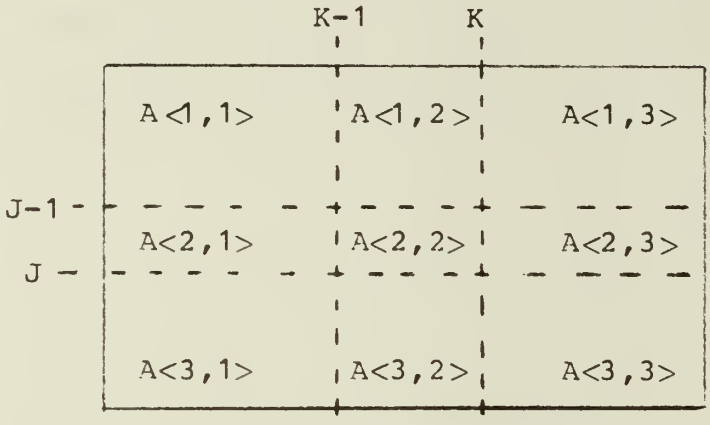


Figure 1 A Partitioned Matrix

Each of the partitioned parts can be given a name such as B = A < 1,1 > , C = A < 3,3 > and used in any subsequent OL/2 statement. In certain instances, the partitioned parts may also be declared to be of type row vector or column vector or scalar. In addition to this type of partitioning, one can give new names to specific predefined parts of arrays. For example, if A is an N by N matrix, one could set L equal to the lower tirangular part of A, or set D equal to the diagonal of A, or both. A complete discussion of dynamic partitioning in OL/2 can be found in Adams [5].

To define these generalized data types at execution time, a structure is created for each OL/2

variable, and is referred to as the root node. Table 1 defines fields for a root node.

Field Designator	Meaning
Name	Name of variable
Attributes	Seven fields which indicate the base, mode, scale, and precision of the variable and whether it is defined, core contained, or a temporary
Dimensionality	Number of dimensions
Modulus	If variable is an array sequence the modulus, else 0
Type	Geometric type of the variable
Row Increment	Number to increment bound to find next row
Diagonal Increment	Number to increment bound to find next diagonal element
Origin	Starting core location
Partition Pointer	Pointer to partitioning control information
Bounds	Upper, lower, and extent for each dimension

Table 1 OL/2 Variable's Root Node

2.2 Types of Expressions

OL/2 includes the usual matrix operations of addition, subtraction, and multiplication for the various data types, as long as the operands are compatible. Exponentiation and scalar multiplication are also defined, as well as the inner product and norm of any vector expression. Also the transpose of any compatible operand

or expression is allowed. Finally, OL/2 allows PL/1 statements along with OL/2 statements in the program.

The generality of the data types and operators make OL/2 a powerful algebraic language. To provide for this generality, a design philosophy was adopted which requires the use of intermediate routines. These routines are called by code generated by the compiler and prepare data for the assembly language calculating routines. The advantages to this design philosophy lie in the tremendous flexibility that one has in implementing current data types and adding other data types later.

3. ARRAY EXPRESSION EVALUATION

The technique for handling array expressions in OL/2 involves two main procedures: 1) parsing the expression into a binary tree and 2) producing the intermediate code to compute the expression.

3.1 Expression Parsing

Array expressions in OL/2 are parsed with respect to the operator precedence relations shown in Table 2. As an expression is parsed, a binary tree of the expression is built with each node containing the information shown in Table 3.

It should be emphasised that a string of scalar operations is not parsed but entered into the name field of a node as a string, and the node is considered as a single variable. For example, Figure 2 shows the expression tree generated by parsing A = ALPHA + BETA x GAMMA x B where A and B are arrays and ALPHA, BETA, and GAMMA are scalars.

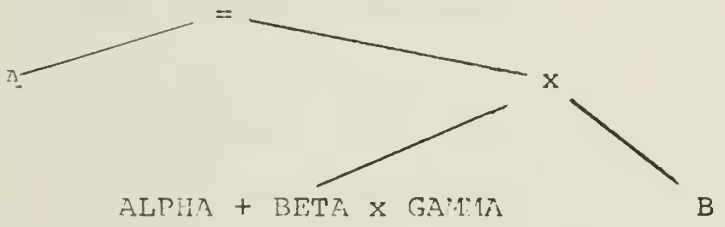


Figure 2 Expression Tree for A = ALPHA+BETAxGAMMAxB

Level	OL/2 Symbol	Meaning
1	'	Transpose
1	+	Uniary Plus
1	-	Uniary Minus (Negation)
2	**	Exponentiation
3	(,)	Inner Product of two Vectors
3	|| ||	Norm of a Vector
4	x + - ** /	Operations with Scalar Operands
5	x	Matrix multiplying Column Vector or Row Vector multiplying Matrix
6	x	Matrix multiplying Matrix or Column Vector multiplying Row Vector
7	/	Matrix or Vector divided by a Scalar
8	x	Scalar multiplying Vector or Vector multiplying Scalar
9	x	Matrix multiplying Scalar or Scalar multiplying Matrix
10	+ -	Array plus or minus Array

Table 2 OL/2 Operator Precedence

	Information at Parse Time	
Node Field	For Variables	For Operators
Name	Name of the pointer to the root node or scalar expression	Null
Sequence number	Number or expression defining which array of a sequence is to be used	Null
Number of dimensions	Number of dimensions of variable	Number of dimensions of expression
Number of temporary variables to free	0	0
Loop end marker	0	0
Type of node	Indicates variable	Type of operator
Type of expression	Type of variable, i.e., vector, matrix, scalar	Type of expression, i.e., vector, matrix, scalar
Right link	Link to right subtree	Link to right subtree
Left link	Link to left subtree	Link to left subtree

Table 3 Tree Node Contents

Figure 3 shows the expression tree for R = AxB + Cx(E+F) (where all variables are compatible matrices) with the name, number of dimensions, and the type of expression fields.

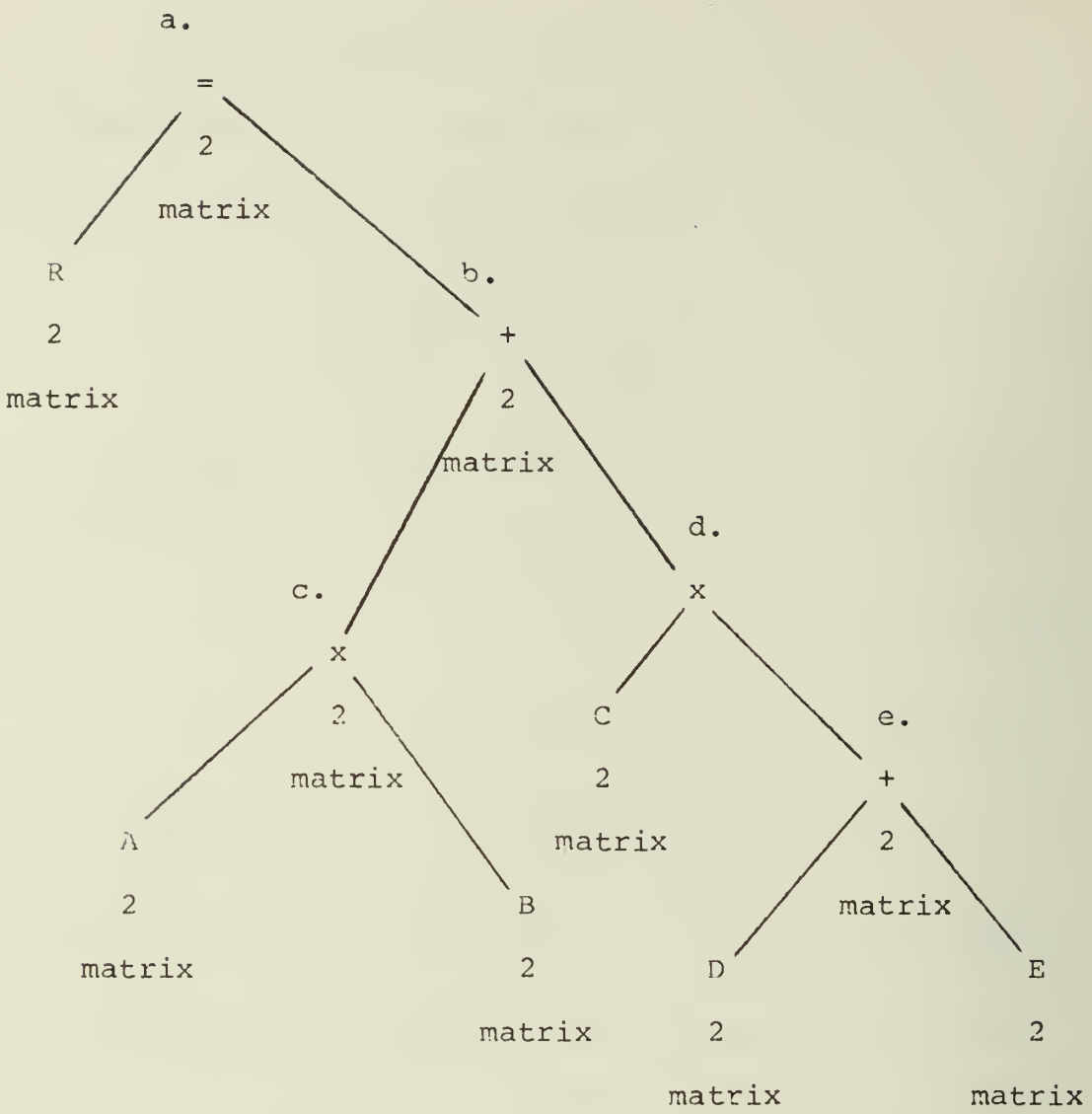


Figure 3 Expression Tree for R = AxB + Cx(D+E)

3.2 Producing Intermediate Code

The procedure for generating intermediate code consists of a section which determines temporary storage requirements and a section which generates the required code.

3.2.1 Determining Temporary Storage Requirements

The general aim of this section is to reduce the temporary storage required to hold intermediate results

computed while evaluating OL/2 array expressions. This is accomplished by organizing the calculation so that row partitions(RP) or column partitions(CP) of array variables are used at each stage and then repeating the calculation via a loop for each RP or CP in the result.

The storage requirements are determined and loop controls are generated as the algorithm traverses the expression tree down toward the terminal nodes. Two stacks are employed during this phase of the operation. The first, called the traverse stack, holds the location of the higher level nodes of the tree passed to reach the current node (the "fathers" of the current node). The second, called the result stack, holds the name of the pointer to the root node and the type of result to be used by each operation of a higher level in the tree than the current node.

The actions taken at each node of the tree are determined by the type of operator in the current node, by the type of nodes on the left and right subtrees, and by the type of result on the top of the result stack. It is expected that the root node of the expression tree will have the type of node field set to an equal operator. If the type of node on the right subtree is not a variable, the type of expression and name from the node on the left subtree are placed on the result stack. The procedure then moves to the right subtree. If the expression being compiled was that shown in Figure 3, after step 1 the result stack would contain name R and type matrix and the

traverse stack would contain the address of node a.

The succeeding actions of this procedure can best be described by dividing it into three cases: 1) left and right subtree are both expressions, 2) left subtree is an expression and right subtree is a variable and 3) left subtree is a variable and right subtree is an expression. The abbreviations in Table 4 will be used in the succeeding subsections to describe the actions of this procedure.

Abbreviation	Meaning
TCV	Place name of a temporary column vector on result stack, set type on result stack to column vector.
TRV	Same as TCV for row vector
TM	Same as TCV for matrix
DRS	Duplicate top of result stack
RLC	Generate control for a loop to contain row partitioning of variables
CLC	Same as RLC for columns

Table 4 Abbreviations

3.2.1.1 Left and Right Subtrees Expressions

Table 5 shows the actions taken when the current node type is a multiply operator.

Type Expression on Right Subtree

Type Expression Left Subtree		Column Vector	Row Vector	Matrix
	Column Vector	Illegal	TCV Walk left	Illegal
	Row Vector	TRV Walk left	Illegal	TRV Walk left
	Matrix	TCV Walk right	Illegal	TM Walk left

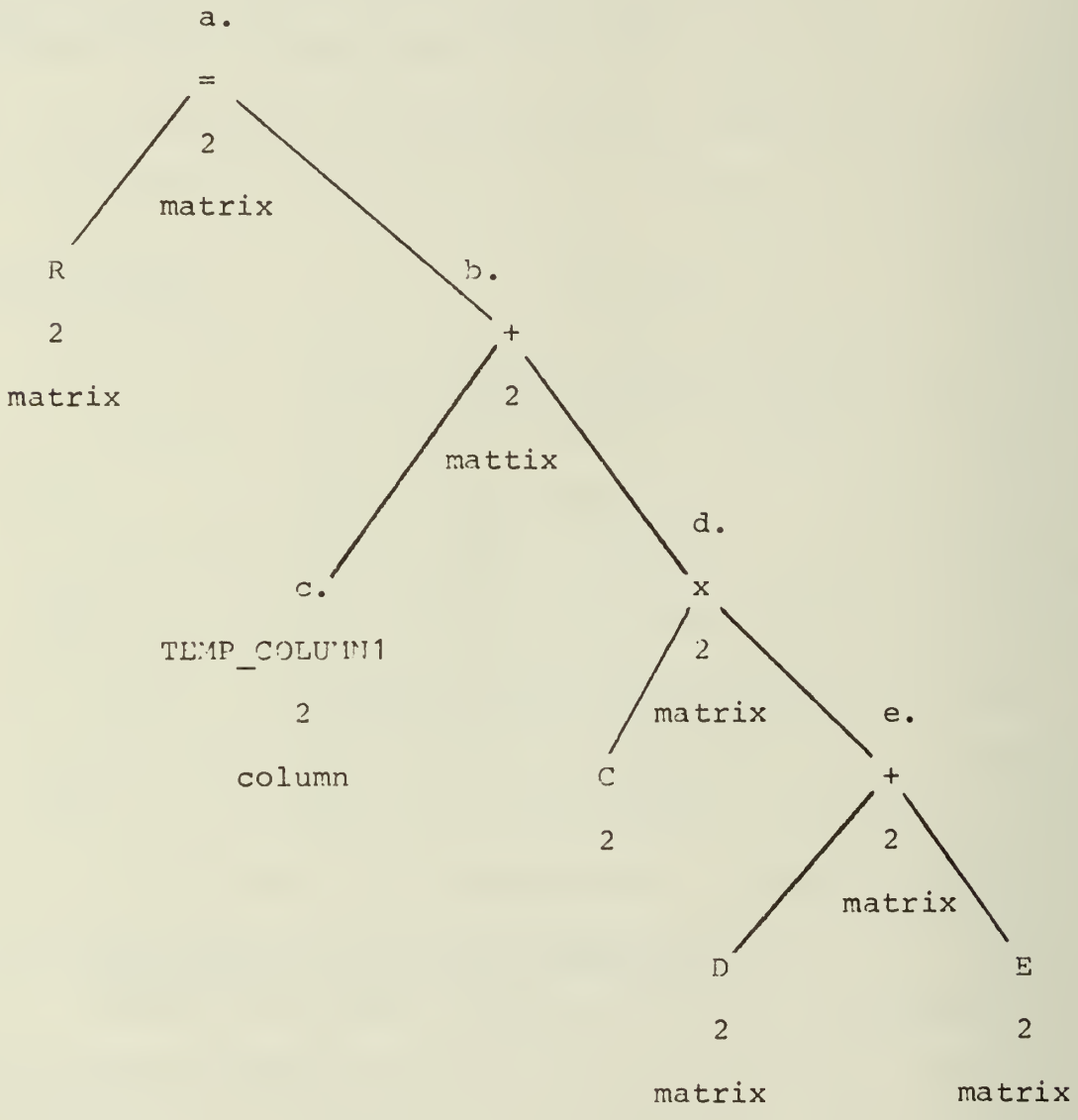
Table 5 Multiply with Both Subtrees Expressions

When the dimensions of the subtrees differ (i.e., matrix x column vector) the algorithm moves to the subtree of lower dimensionality. No loop controls are generated in this case. When the current node type is an addition or subtraction operator, the top of the result stack is duplicated, except in the event that the types of expressions on both subtrees are matrices, and that the type of result on top the result stack is matrix. In this case, control for a loop to contain column partitioning is generated. The calculation of each of the expressions on the subtrees will be organized to compute a column at each stage so that only a temporary column vector will be needed. In either case, the procedure then moves to the left subtree. For the example of Figure 3, control for a column partitioning loop would be generated and we would move to the left subtree. The contents of the stacks at step 2 are shown in Figure 4.

Result Stack		Traverse Stack
TEMP_COLUMN1	column	address node b
R	matrix	address node a

Figure 4 Compiling R = AxB + Cx(D+E), step 2

In our example, code would now be generated to compute TEMP_COLUMN1 = A x Column(B). This is to be discussed fully in section 3.22. It suffices to state here that the expression tree and the stacks would appear as in Figure 5.



Result Stack		Traverse Stack
R	matrix	address node a

Figure 5 Compiling R = AxB + Cx(D+E) step 3

For a node of type inner product operator, a temporary column vector name is placed on the result stack and the algorithm moves to the left subtree. None of the other operators can appear in a node with left and right subtrees both expressions. For division and expontiation the right subtree type expression must be scalar and therefore must be a variable.

3.2.1.2 Left Subtree an Expression and Right Subtree a Variable

After completing processing at the current node, the algorithm will always move to the left subtree to compute the expression. Table 6 defines the actions taken when the current node type is a multiply operator.

Type Expression Left Subtree	Type of Variable on Right Subtree: Scalar	Column Vector	Row Vector	Matrix
Column Vector	DRS	Illegal	TCV	Illegal
Row Vector	DRS	TRV	Illegal	TRV
Matrix	CLC TCV	RLC TRV	Illegal	RLC TRV

Table 6 Multiply with Left Subtree Expression

For the multiply operator there are three combinations which lead to the partitioning of operands. For the cases matrix x scalar and matrix x matrix the type of result on top of the result stack must be matrix and there can not be a partitioning loop in effect. For the cases of scalar multiplication, no additional temporary storage is necessary. Table 7 shows the actions taken for the other types of operators.

Type of Operator	Action
Addition	DRS
Subtraction	DRS
Inner Product	TCV
Division (Matrix Result)	CLC
	TCV
Division (other)	DRS
Exponentiation	TM

Table 7 Actions for Operators with Left Subtree Expression

The case described in this subsection does not appear in the example Figure 3.

3.2.1.3 Left Subtree a Variable and Right Subtree an Expression

In this case, after completing processing at the current node, the algorithm always moves to the right subtree to calculate the expression. Table 8 shows the actions taken when the type of the current node is a multiply operator.

Type of Variable on Right Subtree	Type Expression on Right Subtree: Column Vector	Row Vector	Matrix
Scalar	DRS	DRS	CLC TCV
Column Vector	Illegal	TRV	Illegal
Row Vector	TCV	Illegal	CLC TCV
Matrix	TCV	Illegal	CLC TCV

Table 8 Multiply with Right Subtree Expression

The actions taken in this case are similar to those in the case discussed in the last section. When the current node type is an addition or subtraction operator, the top of the result stack is duplicated, unless the left subtree is a row or column partition, in which case a tempory column vector name is placed on the result stack. For an inner product, or norm, a temporary column vector is also used.

After the last stage of processing, our example was as shown in Figure 5. Since the left subtree is a partitioned column, at step 4 a temporary column vector name is placed on the result stack, and the address of node is placed on the traverse stack. In step 5 another temporary column name is placed on the result stack and the address of node d is placed on the traverse stack. After step 5 the stacks appear as shown in Figure 6.

Result Stack		Traverse Stack
TEMP_COLUMN3	column	address node d
TEMP_COLUMN2	column	address node b
R	matrix	address node a

Figure 6 Compiling R = AxB + Cx(D+E) step 5

3.2.2 Generation of Intermediate Code

This section of the array expression evaluation algorithm generates the intermediate code when the nodes of both subtrees are of type variable. This code consists of statements which call the intermediate routines. The major parameters for these intermediate routines are the pointer to the root node, the type of operand and the row

or column number to use if the operand or result is to be partitioned. The pointer to the root for the operand is available for the name field of the nodes of the tree, as is the type of operand. For the result, this data is available from the result stack. The row or column number for operands to be partitioned is the name of the variable used to control the partitioning loop. The major function of this part of the program is to determine when to partition an operand or result. This determination is made by utilizing the type of result from the result stack and the type of expression fields from the left and right subtree nodes. Table 9 shows the parameters generated for the multiply operation. The general format for this table is:

Type of Result

Type left operand	Partition type
Type right operand	Partition type
Type of result	Partition type

For addition and subtraction, the type of result from the result stack, the type of the operands, and the number of dimensions of the operands determine whether an operand is to be partitioned. The number of dimensions' field in the tree node is used to distinguish between operands which are "true" vectors and vectors which arise from operands being partitioned by this evaluation technique. If the dimension is one then the variable is a "true" vector and will not be partitioned; otherwise,

Left \ Right	Scalar	Column Vector	Row Vector	Matrix
Scalar	Illegal	Column Scalar None Column None Column None		Column Scalar None Column Col Column None
			Row Scalar None Row None Row None	Row Scalar None Row Row Row None
		Matrix Scalar None Column None Column Col	Matrix Scalar None Row None Row Row	Matrix Scalar None Matrix None Matrix None
Column Vector	Column Vector Column None Scalar None Column None	Illegal	Column Vector Matrix Column Col Column	Illegal
			Row Vector Row Row Matrix None Row None	
	Matrix Column None Scalar None Column Col		Matrix Matrix None Matrix None Matrix None	
Row Vector	Row Vector Row None Scalar None Row None	Column Vector Row None Column None Row Row	Illegal	Row Vector Row None Matrix None Row None
	Matrix Row None Scalar None Row Row			Matrix Row None Matrix None Row Row
Matrix	Column Vector Column Col Scalar None Column None	Column Vector Matrix None Column None Column None	Illegal	Column Vector Matrix None Column Col Column None
	Row Vector Row Row Scalar None Row None			Row Vector Row Row Matrix None Row None
	Matrix Matrix None Scalar None Matrix None	Matrix Matrix None Column None Column Col		Matrix Matrix None Matrix None Matrix None

Table 9 Operand Partitioning for Multiply

the operand may be partitioned. The result of the operation will be partitioned if it is of type matrix and the operands are of type vector.

The intermediate code for computing inner products and norm consists of a scalar assignment statement. The computation is accomplished by a function subroutine and is assigned to a temporary scalar variable.

For the division operator, the parameters for the left operand and result are determined in the same manner as for addition. The right operand is always a scalar.

After the intermediate code is generated, the type of node field in the current node is changed to variable and the name field is changed to the name on the top of the result stack. The result stack is then popped, and any temporary storage used on the level of the tree below is marked to be freed. Next, the algorithm moves to the node on the top of the traverse stack and pops the traverse stack.

We may now complete our example. The algorithm is now at node e and finds both subtrees are variables. Since the type of result is a column (see Figure 6) and the number of dimensions of both variables is two (see Figure 5), both variables must be partitioned into columns. Codezis then generaged to add a column of D to a column of E each time through the loop. The name field of node e is changed to the name on the top of the result stack,

and the result stack is popped. The algorithm moves to the node on the top of the traverse stack and popszthis stack. At this stage of the process the stacks and the tree are as shown in Figure 7.

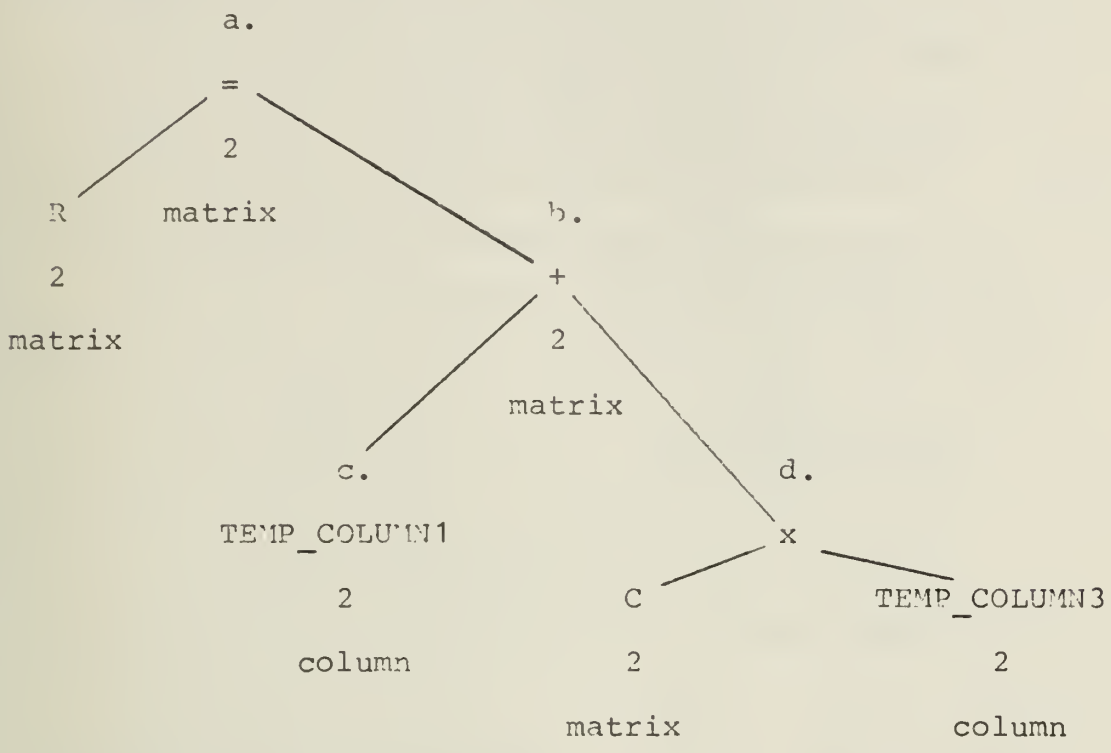


Figure 7 Compiling R = AxB + Cx(D+E) step 6

The algorithm then follows the actions outlined in Table 3.8; it multiplies the matrix C times the temporary column vector and stores the result in TEMP_COLUMN2. Node d is then changed to this variable and the algorithm pops the stacks and moves to node b. Here the two temporary column vectors are added and the result stored in the proper column partition of R.

4. IMPLEMENTATION

The OL/2 language has been implemented by using the TACOS compiler-compiler developed by Gaffney [7]. The syntax of the source language is specified in a modified BNF form, called IBNF. Productions are written as Phrase class ::= definition. For example, TERM ::= FACTOR TERM x FACTOR. To avoid left recursive phrase class definitions, such as the example above, TACOS uses repetition characters +, *, and ? which are defined in Table 10.

BNF	IBNF
A ::= B A B	A ::= B +
A ::= empty A B	A ::= B *
A ::= empty B	A ::= B ?

Table 10 BNF vs IBNF Definitions

There are several special notations used to define a language for TACOS: *I causes an attempt to recognize an identifier (legal to PL/1), while *N causes an attempt to recognize an unsigned intger. The last important notation for IBNF is #n, where n is an integer. This causes a transfer to a separately compiled semantic action routine defined by the number n. PL/1 is used as the semantic language for TACOS, so all semantic action routines for a language are placed in a recursive PL/1 procedure. This procedure is given n as a parameter, and it branches to the proper routine via a label array. The semantic action routines and TACOS communicate by sharing the set of common data fields shown in Table 11.

Data Field	Usage
TEMPCONST	location of integer recognized by *N notation
TEMPIDENT	location of identifier recognized by *I notation
CHAR	source language input area
OK	success (1) or failure (0) of phrase class indicator
INP	subscript of CHAR pointing to current input character

Table 11 Common Data Fields

There are also two external procedures used by the semantic action routines: SCAN_UNTIL_PASS which scans the input string (CHAR) until one of the characters given as a parameter is found and increments the value of INP accordingly; and SCAN_UNTIL_KEEP which scans similarly to SCAN_UNTIL_PASS but also places the string passed over into TEMPSTRING.

4.1 OL/2 Syntax and Semantics

Our purpose in this section is to describe some of the more important semantic action routines associated with array expressions. The reader is referred to Appendices A and B for listings of the syntax and action routines for expressions.

Other sections of the OL/2 compiler generate data which is used during the evaluation of array expressions. When a variable is recognized in an OL/2 declaration, its name, type, and dimensionality are entered into the OL/2 identifier talbe. During this process, a tree node (Table 3) is also built. Later if this variable

appears in an assignment statement this node will be used in constructing the tree [6]. Similar action is also taken when a partitioned part of an array is given a name [5].

Table 12 details the major semantic action routines and their function.

Name	Function	Called by	Calls
ACTION_5	Print Source Statement	TACOS	
ACTION_8	Determines if assignment statement follows, initialization	TACOS	
ACTION_12	Builds Expression Tree	TACOS, #16, #17 #18, #19, #21 #22, #29, #34 #42, #44	
ACTION_16	Determine Multiplication Precedence	TACOS	#12
ACTION_17	Determine Multiplication Precedence	TACOS	#12
ACTION_20	Determine if parenthesized expression or inner product	TACOS	
ACTION_23	Determine if identifier is OL/2 variable	TACOS	
ACTION_24	Mark beginning of scalar String	TACOS	

Name	Function	Called by	Calls
ACTION_25	Place single scalar string in name field of tree node	TACOS	
ACTION_27	Determine if identifier is OL/2 subscripted variable, or function entry	TACOS	
ACTION_31	Determine if identifier is OL/2 scalar	TACOS	
ACTION_34	Assign norm, inner product to temporary scalar variable	TACOS, #36	#12 CODER
ACTION_39	Determine type of identifier	TACOS	
ACTION_42	Set node for = operator in tree and call for code generation	TACOS	#12 CODER
ACTION_48	Insure what follows is an inner product	TACOS	
ACTION_104	Obtain the string identifying the sequence number	TACOS	
ACTION_107	Output statement not recognized as OL/2 or PL/1	TACOS	
ACTION_113	Obtain string identifying partition part	TACOS	
ACTION_140	Output PL/1 statements	TACOS	

Table 12 Semantic Action Routine Functions

The semantic routines use a stack, called the parsing stack, to aid in building the expression tree. This stack has two entries: the first is a pointer to a subtree which is an operand of some operator (SUBTREE_PTR); and the second is the type of expression of this subtree (SUBTREE_TYPE).

Example 1, R = A + B;, where all the variables are arrays, will be used to illustrate the major steps in the compilation of assignment statements. In the first step for recognizing an ASSIGNMENT_STATEMENT, ACTION_8 insures that an equal operator is present in the statement. Secondly, we attempt to recognize OL2_LEFT_HAND_SIDE by finding an OL2_IDENTIFIER. The *I notation places the identifier R in TEMPIDENT; then ACTION_23 searches the OL/2 identifier table and determines that R is an OL/2 identifier. ACTION_23 then places the pointer to the tree node and type for R on the parsing stack. The OL2_IDENTIFIER phrase class has been found and since none of the other constructs following it in OL2_LEFT_HAND_SIDE are requiredzthe OL2_LEFT_ HAND_SIDE phrase class has been recognized. We next find the eugal operator and start searching for OL2_ARITHMETIC_EXPRESSION. Starting with OL2_TERM we search down the svntax until OL2_IDENTIFIER is found. ACTION_23 places the tree node pointer and type for A on the parsing stack. Moving back up the syntax, we discover OL2_TERM has been found, and then we recognize the + operator. Again we move down the syntax, finding and stacking the identifier B. Next ACTION_10 would set

the CURRENT_OP field to addition and we go to ACTION_12. At this point the parsing stack is as shown in Figure 8.

SUBTREE PTR	SUBTREE TYPE
B	matrix
A	matrix
R	matrix

Figure 8 Parsing Stack for R = A + B after All Operands Recognized

The semantic routine ACTION_12 actually links the nodes of the expression tree. ACTION_12 is entered after an operator and its operand(s) have been recognized. Upon entry to ACTION_12, a tree node for the operator is allocated and the type of node field set to CURRENT_OP. It next branches to the proper subsection for the current type of operator where some simple error checking is accomplished and the type of expression and number of dimensions fields are entered. If the operator was binary, the pointer from the top of the parsing stack is placed in the right link field (RLINK) and the next level pointer in the left link field (LLINK). The pointer to the operator node then replaces the top two levels of the stack. For uniary operators, the pointer in the top of the stack is placed in RLINK and the pointer to the operator node replaces the top level. After execution of ACTION_12 the parsing stack for our operator would appear as in Figure 9.

SUBTREE PTR	SUBTREE TYPE
E1	matrix
R	matrix

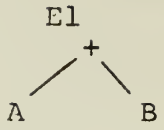


Figure 9 Parsing Stack for R = A + B after Addition Operator Processed

The OL2_ARITHMETIC_EXPRESSION has now been recognized. ACTION_42 next sets CURRENT_OP to an equal operator and calls ACTION_12 which results in a complete expression tree. ACTION_42 now calls the code generating program (CODER), the semicolon is recognized, and the processing of this statement is complete.

The operator precedence table (Table 2) shows that the interpretation of the multiply operator is dependent upon the context in which it appears. ACTION_16 and ACTION_17 determine this context by comparing the type of expression parts of the parsing stack and using a precedence table. Consider example 2, Y = A x Z x ALPHA, where Y and Z are column vectors, A is a matrix and ALPHA is a scalar. This expression is parsed similarly to the previous example until the first multiply operator is reached; at this point ACTION_13 places a marker on the parsing stack to indicate the start of multiply operations. Next multiply operator and the identifier Z are recognized, CURRENT_OP is set to multiply, and ACTION_16 entered. At this point, the parsing stack appears as shown in Figure 10.

SUBTREE PTR	SUBTREE TYPE
Z	column vector
A	matrix
	MARKER
Y	column vector

Figure 10 Parsing Stack for R = AxZx ALPHA after First Multiply Is Recognized

ACTION_16 determines that a matrix times a column vector has the highest precedence and calls ACTION_12 to build that section of the tree. Next the scalar ALPHA is placed in the stack, and ACTION_16 determines that a vector x scalar is not of sufficiently high precedence to construct this part of the tree, therefore, ACTION_12 is not called. ACTION_17 is entered with the parsing stack as shown in Figure 11.

SUBTREE PTR	SUBTREE TYPE
ALPHA	scalar
E1	column vector
	MARKER
Y	column vector

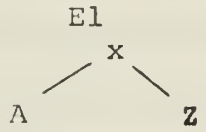


Figure 11 Parsing Stack for R = AxZx ALPHA upon Entry to ACTION_17

ACTION_17 then unstacks all operands in the parsing stack down to the marker by calling ACTION_12 for each pair and removes the marker. The parsing stack would appear as shown in Figure 12 after execution of ACTION_17.

SUBTREE PTR	SUBTREE TYPE
E2	column vector
Y	column vector

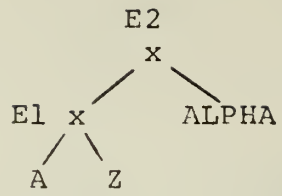


Figure 12 Parsing Stack for R = AxZx ALPHA after Execution of ACTION_17

The processing of this statement then continues as in example 1. In Section 3, it was noted that a string of scalar operations is not parsed, but passed as a string to the PL/1 compiler. Consider the example from Section 3, as example 3: A = ALPHA + BETA x GAMMA x B; where A and B are matrices and ALPHA, BETA, GAMMA are scalars. The identifier A and the = operator are recognized as in our previous examples. In attempting to recognize OL2_ARITHMETIC_EXPRESSION, we will eventually attempt to recognize the SCALAR_EXP phrase class. ACTION_24 places a marker on the parsing stack to indicate the start of a scalar string and saves the current location in the input string (INP). The scalar string is passed over, as it is recognized by the syntax. When the end of the string is recognized, ACTION_25 takes the string, the saved input location to the current value of INP, and places it in the name field of a tree node; and then places the node on the parsing stack. ACTION_25 subsequently removes the marker from the stack and the rest of the statement is parsed as in our previous examples.

It is also possible to build and stack several nodes for a scalar string if the string includes an OL/2

variable with subscripts, an inner product, or a norm. As example 4 we will extend example 3 to include the inner product of two vectors X and Y; A = ALPHA + BETA x GAMMAx (X,Y) x B. The scalar string up to the inner product is recognized as in the previous example. ACTION_48, after determining that an inner product follows, places this string on the parsing stack via a call to ACTION_37 (Figure 13). Next, the operands for the inner product are recognized and placed on the parsing stack (Figure 14), CURRENT_OP is set to inner product, and ACTION_12 is called. An expression tree is then created assigning the inner product to a tempory scalar variable (Figure 15) and CODER is called by ACTION_34. The name of the temporary variable is then placed in a node on the parsing stack (Figure 16) ACTION_25 now takes the strings from all nodes above the marker, and places them in a single node and removes the marker (Figure 17). The identifier B is then recognized and processing would continue as in example 3.

SUBTREE PTR	SUBTREE TYPE
ALPHA+BETAxGAMMAx	scalar
	MARKER
	matrix

Figure 13 Parsing Stack for R = ALPHA+BETAxGAMMAx(X,Y)xB

stage 1

SUBTREE PTR	SUBTREE TYPE
Y	column vector
X	column vector
ALPHA+BETAxGAMMAx	scalar
	MARKER
A	matrix

Figure 14 Parsing Stack for R = ALPHA+BETAxGAMMAx(X,Y)xB

stage 2

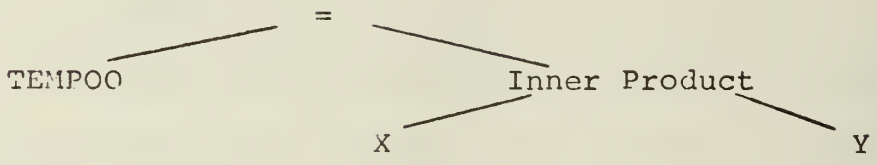


Figure 15 Expression Tree for Inner Product

SUBTREE PTR	SUBTREE TYPE
TEMPOO	scalar
ALPHA+BETAxGAMMAx	scalar
	MARKER
A	matrix

Figure 16 Parsing Stack for R = ALPHA+BETAxGAMMAx(X,Y)xB

stage 3

SUBTREE PTR	SUBTREE TYPE
ALPHA+BETAxGAMMAxTEMPOO	scalar
A	matrix

Figure 17 Parsing Stack for R = ALPHA+BETAxGAMMAx(X,Y)xB

stage 4

The actions described for inner product are identical to those accomplished for a norm. For an OL/2 variable with subscripts or an OL/2 variable declared to be a scalar, a function routine call would be placed on the stack by ACTION_27 instead of the temporary scalar variable; otherwise the process would be identical.

These are the operations of the major semantic action routines. Appendix B gives the listing of all the semantic routines for building the expression tree.

4.2 OL/2 Code Generator

The OL/2 code generator (CODER) has been implemented in PL/1 as an external procedure. The reader is referred to Appendix C for a complete listing of this program. CODER is called by semantic routines ACTION_34 or ACTION_42 when an expression tree has been completed, sending the root of the expression tree as the major parameter.

Conventions have been developed for code generated by the OL/2 compiler to facilitate module interactions and readability. A summary of these which effect array expression are given in Table 13.

Convention	Meaning
$ (variable)	Pointer to the root node for (variable)
$TEMP(n1)(n2)	Pointer to the root node for temporary variable where n1 is dimension and n2 a sequence number
@OL (routine name)	an OL/2 intermediate routine
# (variable)	a non array temporary variable

Table 13 OL/2 Code Conventions

As explained in section 3, the temporary storage and partitioning control requirements are determined by the type of subtrees of an operator node. A label array, CODER_OPER, is used to switch to the proper subsection of the program for each combination of operator and subtree types. There are eight subsections of CODER which place a pointer to root node name (PTR_TO_RESULT_NODE) and type of result (RESULT_TYPE) on the result stack, two each for

column vectors, row vectors, matrices, and duplicating the top of the result stack. One of each pair moves to the left subtree (WALK_LEFT) and one moves to the right subtree (WALK_RIGHT). The operation of each of the eight subsections is similar. First, a position on top of the result stack is allocated; secondly, the next legal root node pointer name for the temporary variable required and its type are placed on the stack. In the third step, the current node is marked as requiring the temporary variable by setting the #TEMP_TO_FREE field in the tree node to one. The program then moves to the proper subtree.

There are two operator types which are treated as special cases: transpose and uniary minus (negation). These two operators do not truly create a new result but only modify an operand for another operator. For this reason, if one of the subtrees of an operator node is a transpose or uniary minus operator node, nothing is placed on the result stack at this point and the program simply moves to the subtree. If a transpose operator node has an expression as a subtree, a name and type to hold the result of that expression will be placed on the result stack. For a uniary minus operator node, the top of the result stack is duplicated.

When control for the partitioning of operands is required one of three subsections of CODER is executed. one controls row partitioning and the other two control column partitioning. Operand partitioning is controlled by generating a PL/1 DO loop which will increment a control

variable (#ROW or #COL) from the lower to the upper bound found in the root node named on the top of the result stack. Next, the current node is marked as having initiated an operand partition loop by setting #CEND_STATEMENTS or #REND_STATEMENTS to one. In addition, an indicator called LOOP_DEPTH is set to one to indicate that operand partitioning is in effect. For the example given in section 3, (Figure 3) when node b is reached the following code would be generated:

```
DO #COL1 = $R -> #LOWER(2) TO $R -> #UPPER(2);
```

The generation of code to calculate results is accomplished when an operator node is reached and both subtrees are variables. The generated code has been called intermediate code since it is in the form of calls to intermediate routines and not directly to the assembly language computation programs. For binary operators, CODER generates the seven parameters shown in Table 14 for each operand. For the result, only the parameters name, sequence number, type, and row or column number are generated. The first five of the parameters for operands and the first two for the result are assembled by the internal procedure SET_VARIALBES. The name and sequence number parameters are taken from the tree nodes for the operands and the result stack for the result. The save indicator is always set to one and appears only to be compatible with another code generating program. The transpose and negation parameters are taken from indicators set for each level of the expression tree. The indicators are set when a

transpose or uniary minus operator node is encountered with a variable as its subtree.

Parameter	Contents
Name	Pointer to root node
Sequence Number	Expression indicating which array of a sequence to use; 0 is no sequence
Transpose	1 if operand to be transposed before use, 0 otherwise
Save	0 if storage for this operand to be released, 1 if to be saved
Negate	1 if operand to be negated before use, 0 otherwise
Type	0-scalar, 2-column vector 3-row vector, 4-matrix
Row or Column Number	The control variable of partition loop or 0

Table 14 Operand Parameters

The last two parameters for operands and results are entered by two major subsections of CODER. The first, SET_ROW_OR_COL, determines these parameters for addition, subtraction, and division, and operates as described in section 3. The second subsection, CODER_MULT_VAR_VAR, implements Table 3.8 in section 3.

We will take our example from section 3 (Figure 3.2) and assume we are at node c with stacks as shown in Figure 3.3. The type of result is column vector and the type of both variables is matrix so we partition the second operand by columns and the resultant code would be

```
CALL@OLMULT($A,0,0,1,0,4,0,$B,0,0,1,0,2,#COL1,$TEMP10,0,2,0);
```

After producing the code, the type of the current node ($TYPE_CODE) is set to variable and the name and type of expression in the node are changed to those on top of the result stack. The WALK_UP subsection of CODER is then executed.

In WALK_UP, the current node is examined to determine if it caused the initiation of an operand partitioning loop. If it did, the PL/1 END statement is generated and the indicator LOOP_DEPTH is set to zero. Next, code is generated to release any temporary storage used to hold intermediate results for lower levels of the tree. The current node is set to the top of the traverse stack and the next action required is determined.

We will pick up our example from section 3 with e as the current node and the stacks as shown in Figure 6. Both operands are matrices and a column result is required, so both operands are partitioned by columns and the generated code is

```
CALL@OLADD($D,0,0,1,0,2,#COL1,$E,0,0,1,0,2,#COL1,$TEMP12,0,2,0);
```

The name field of node e is set to $TEMP12 and its type to column vector. Next, at node d, we have a matrix times a column vector with a column result, so no partitioning is required. The code generated is

```
CALL@OLMULT($C,0,0,1,0,4,0,$TEMP12,0,0,1,0,2,0,$TEMP11,0,2,0);
```

We moved to node b through WALK_UP and could release the storage for the temporary $TEMP12 at this point, but we do not, since we will need it each time through our operand partitioning loop. The code to realse temporary storage

is saved until the end of an operand partitioning loop. At node b, both operands are column vectors and the result is a matrix, so the result is partitioned and the code would be

```
CALL@OLADD($TEMP10,0,0,1,0,2,0,$TEMP11,0,0,1,2,0,$R,0,2,#COL1);
```

As we move to node through a WALK_UP, the END statement for the loop is generated, as is the code to release all temporary storage.

CODER saves the lines of code it generates (in OUTLINE) and only outputs them when an operand partition loop is not in effect (LOOP_DEPTH=0). This is done because it is possible for an assignment statement to be written so as to necessitate the addition or multiplication of a full matrix after an operand partitioning loop control has been generated. An example of such a statement is R = ALPHAx((AxBx)CxD)) whose expression tree is shown in Figure 18.

The operations of CODER would be: 1) at node b, a loop to control operand partitioning by column would be generated, 2) at node c, the name for a temporary matrix would be placed on the result stack, 3) and at node d the code to multiply the full matrices A and B would be generated. Obviously one does not want to do this operation inside a loop, so in this situation the code would be output and not saved.

Appendix D contains a sample OL/2 program and the intermediate code generated by CODER.

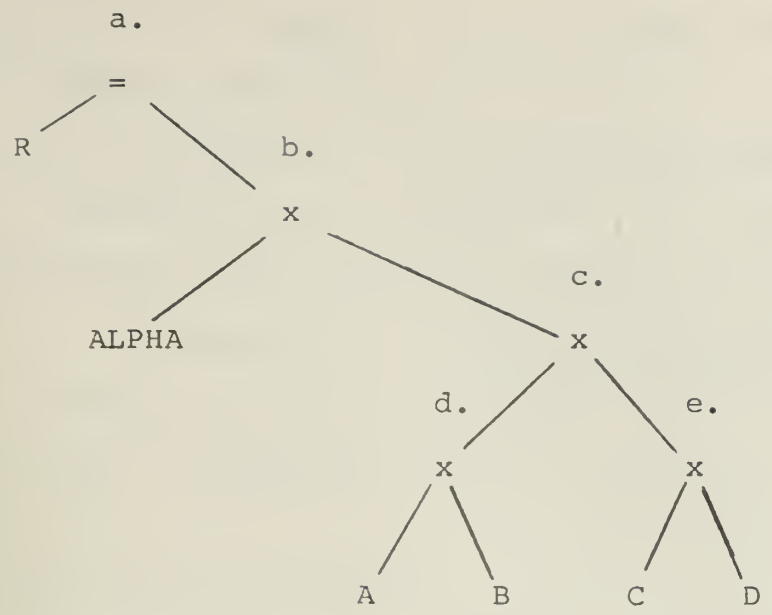


Figure 18 Expression Tree for R = ALPHAx((AxB)x(CxD))

4.3 Intermediate Routines

The intermediate routines perform the vital function of preparing the parameters needed by the assembly language calculating routines from the data generated by CODER. There are eleven intermediate routines to which calls are generated by CODER; several of them actually call other intermediate routines and have been used to improve readability of the generated code. Table 15 contains a list of these programs.

The intermediate routines draw mainly on the information contained in the root node (Table 1) for each operand. The major parameters passed to the assembly language routines are the core location (origin), bounds, geometric type, and increments of each operand. For operands which are not partitioned or are not part of a sequence, the data is taken directly from the root node. For operands which are one of a sequence, the operand

origin is computed by adding the array length times the sequence number to the origin of the first array of the sequence. For partitioned operands the intermediate routines must compute the origin, bounds, and increments for the particular row or column required. This computation is accomplished in the same manner as for dynamic partitioning of variables [5]. For transposed operands the bounds are interchanged, and also the geometric type code is often changed.

In addition to preparing the parameters for the assembly language routines, the intermediate programs allocate the storage and initialize a root node for temporary variables. They compute the minimum storage required depending on the geometric types of the operand and then call a storage allocation routine. They also perform error detection, especially for computability of operands.

Operation	Name	Function
Addition	@OLADD	Calls assembly language addition routine
Subtraction	@OLSUB	Resets negation parameter of second operand and calls @OLADD
Multiplication	@OLMULT	If operands are both arrays, calls assembly language multiplication routine. If one operand is scalar, calls assembly language scalar multiplication routine
Division	@OLDIVD	Takes the recriprical of the second operand and calls @OLMULT
Inner Product	@OLIPRD	A function routine which returns the inner product value. Calls @OLMULT
Norm	@OLNORM	A function routine which returns the value of the norm. Calls norm assembly language routine
Assignment	@OLASGN	Assign one OL/2 variable to another by calling @OLADD with one null operand.
OL/2 Scalar	@OLSCAL	A function routine to return the value of an OL/2 scalar
Subscripted OL/2	@OLELEM	A function routine to return the value of a subscripted OL/2 variable
Exponentiation	@OLEXPN	Call @OLMULT a variable number of times depending on exponent
Free Storage	@OLFSTR	Releases storage for temporary variables

Table 15 OL/2 Intermediate Routines

LIST OF REFERENCES

[1] B. A. Galler and A. J. Perlis, "Compiling Matrix Operations", CACM, 5, (Dec.,1962), pp590-594.

[2] J. Reinfelds, "An Implementation of Automatic Array Arithmetic by a Generalized Push-Down Stack", Interactive System for Experimental Applied Mathematics, Academic Press, New York, (1968), pp441-422.

[3] R. A. Wagner, "Some Techniques for Algorithm Optimization with Application to Matrix Arithmetic Expressions", PhD Thesis, Carnegie-Mellon University, Pittsburgh, (1968).

[4] B. A. Galler and A. J. Perlis, A View of Programming Languages, Addison-Wesley, Reading, (1970), pp231-268.

[5] H. C. Adams, "Dynamic Partitioning in the Array Language OL/2", M.S. Thesis, Report No. 421, Department of Computer Science, University of Illinois, Urbana, (1971).

[6] J. R. Phillips, "The Structure and Design Philosophy of OL/2 -An Array Language", Report No. 420, Department of Computer Science, University of Illinois, Urbana, (to be published).

[7] J. L. Gaffney Jr, :TACOS: A Table Driven Compiler-Compiler System", M.S. Thesis, Report No. 325 Department of Computer Science, University of Illinois, Urbana, (1969).

APPENDIX A

OL/2 SYNTAX FOR ASSIGNMENT STATEMENTS

```
<TEST_PROGRAM> ::= ( <PL1_STATEMENT> | <#5> (<OL2_STATEMENT> | <#107>
                   ))*   ;


<PL1_STATEMENT> ::= '%' <#140>;


<OL2_STATEMENT> ::= <OTHER_TYPES_OL2_STATEMENTS> |
                      <ASSIGNMENT_STATEMENT>  ;


<ASSIGNMENT_STATEMENT> ::= <#8> <OL2_LEFT_HAND_SIDE> ( ','
           <OL2_LEFT_HAND_SIDE> )* ( '=' | '<-' )  <#41>
           ( ( '@' | '"NULL"' ) <#46> | <OL2_ARITHMETIC_EXPRESSION>
           <#42> ) ';'    ;


<OL2_LEFT_HAND_SIDE> ::= <OL2_IDENTIFIER>
           ( '|' <#104> '|' <#22> )? ( ( '<' <#113> '>' )+ <#44> )?
         | (<#24> <REFERENCE> <#25> | <#40> )    ;



<OL2_ARITHMETIC_EXPRESSION> ::= <OL2_TERM> ( ( '+' <OL2_TERM> <#10>
           | '-' <OL2_TERM> <#11> ) <#12> )*                          ;


<OL2_TERM> ::= <OL2_DIVIDE> <#13> ( '*' <OL2_DIVIDE> <#14> <#16>
           )* <#17>                                                   ;


<OL2_DIVIDE> ::= <OL2_FACTOR> ( '/' (( '+' )? <EXPRESSION_UNIT> | '-'
           <EXPRESSION_UNIT> <#19> | ( <#24> <BASICS> <#25> | <#40> )
            | <MODIFIED_OL2_IDENTIFIER> ) <#15> <#16> )*               ;


<OL2_FACTOR> ::= <OL2_PRIMARY> ( '**' <OL2_EXTRA> <#18> )?            ;


<OL2_EXTRA> ::= ( ( '+' )? <EXPRESSION_UNIT> | '-' <EXPRESSION_UNIT>
           <#19> | ( <#24> <BASICS> <#25> | <#40> ) ) ( '**'
           <OL2_EXTRA> <#18> ) ?                                      ;


<OL2_PRIMARY> ::= ( ( '+' )? <EXPRESSION_UNIT> | '-' <EXPRESSION_UNIT>
           <#19> ) | <SCALAR_EXP> | <MODIFIED_OL2_IDENTIFIER>         ;


<EXPRESSION_UNIT> ::= '(' <#20> <OL2_ARITHMETIC_EXPRESSION> ')'
```

```
          ( '''' <#21> )?                                                    ;

<MODIFIED_OL2_IDENTIFIER> ::= ( '-' <OL2_IDENTIFIER> <#19> | ( '+' )?
          <OL2_IDENTIFIER> ) ( '|' <#104> '|' <#22>)?
          ( ( '<' <#113> '>' )+ <#44> )? ( '''' <#21> )?    ;

<OL2_IDENTIFIER> ::= <*I> <#23>                                              ;

<SCALAR_EXP> ::= <#24> <PL1_AND_OL2_SCALARS> <#25> | <#40>                   ;

<PL1_AND_OL2_SCALARS> ::= <TERM> ( ( '+' | '-' ) <TERM> )*                   ;

<TERM> ::= <FACTOR> ( ( '*' | '/' ) <FACTOR> )*                              ;

<FACTOR> ::= <PRIMITIVE> ( '**' <FACTOR> )*                                  ;

<PRIMITIVE> ::= ( '+' | '-' )? <BASICS>                                      ;

<BASICS> ::= '(' <#20> <PL1_AND_OL2_SCALARS> ')' | <NORM> |
          <INNER_PRODUCT> | <REFERENCE> | <CONSTANT>                         ;

<REFERENCE> ::= <BASIC_REF> ( '->' <BASIC_REF> )*                            ;

<BASIC_REF> ::= <UNQUAL> ( '.' <UNQUAL> )*                                   ;

<UNQUAL> ::= <#26> <*I> <#39> ( '(' <#27>
          <OL2_ARITHMETIC_EXPRESSION> <#28> ( ','
          <OL2_ARITHMETIC_EXPRESSION> <#28> <#29> )* ')' <#30>
           | <#31> ) <#38>                                                   ;

<CONSTANT> ::= <#32> ( <*N> )? ( '.' )? ( <*N> )? <#33>
          ( 'E' ( '+' | '-' ) <*N> ( 'I' )? )?                               ;

<NORM> ::= '||' <#37> <OL2_ARITHMETIC_EXPRESSION> '||' <#34>                 ;

<INNER_PRODUCT> ::= '(' <#48> <OL2_ARITHMETIC_EXPRESSION> <#35>
          ',' <OL2_ARITHMETIC_EXPRESSION> <#35> ')' <#36>                    ;
```

APPENDIX B

OL/2 SEMANTIC ACTION ROUTINES FOR ASSIGNMENT STATEMENTS AND ARITHMETIC EXPRESSIONS

```
ACT:  /*  OL/2 SEMANTIC ACTION ROUTINE PROCEDURE  */
      PROCEDURE(WHICH_ACTION_NUM) RECURSIVE;
       DCL WHICH_ACTION_NUM FIXED BIN (31,0) ,
         1 BRIDGE EXTERNAL,
         2 GOCONDITION FIXED BIN (31,0) ,
         2 TEMPCONST VARYING CHARACTER(15),
         2 TEMPIDENT VARYING CHARACTER(32),
         2 TEMPSTRING VARYING CHARACTER(100),
           CHAR(32767) CHAR(1) EXTERNAL CONTROLLED,
           (OK , INP) FIXED BINARY (31,0) EXTERNAL ,
           CARDNUM EXTERNAL ENTRY RETURNS(FIXED BINARY (31,0)) ,
           CARDCOL EXTERNAL ENTRY RETURNS(FIXED BINARY (31,0)) ,
           ACTION(0:200) LABEL STATIC;


 ACTION_5:  /* PRINT SOURCE STATEMENT */
            CALL SKIP_AND_OUTPUT;
            I=INP;
            DO WHILE (CHAR(I)=' ');
                I=I+1;        END;
            INP = I  ;
            IF STATEMENT_PRINTED = YES THEN GO TO RETURN_TO_PARSER;
            OUTPUT_BUFFER='/* ';
      ACT5CNUM:
            L = ((INP-1)/72)+1    ;
            DO WHILE ( L =  ((INP-1)/72)+1 )  ;
                IF CHAR(INP) = ' ' THEN GO TO SET_OK_ZERO_AND_RETURN;
                IF CHAR(INP) = ';' THEN GO TO ACT5END;
                OUTPUT_BUFFER=OUTPUT_BUFFER || CHAR(INP);
                INP=INP+1;
                END;
            CALL SKIP_AND_OUTPUT;
            GO TO ACT5CNUM;
      ACT5END:
            OUTPUT_BUFFER=OUTPUT_BUFFER || '; */'  ;
            CALL SKIP_AND_OUTPUT;
            INP=I;
            CALL SKIP_AND_OUTPUT;
            GO TO RETURN_TO_PARSER;

 ACTION_8:  /* INITIALIZE TO PROCESS AN ASSIGNMENT STATEMENT          */
            /* DO A SEMANTIC SCAN TEST FOR AN ASSIGNMENT STATEMENT   */
            I = INP ;
            CALL SCAN_UNTIL_PASS( ';' , '=' , '<-' ) ;
            J = INP ;
            INP = I ;
            IF CHAR(J) = ';' THEN GO TO SET_OK_ZERO_AND_RETURN ;
            STK_PTR = 0 ;
            SCALAR_TEMP_VARIABLE_#=0;
```

```
            VECTOR_TEMP_VARIABLE_#=0;
            MATRIX_TEMP_VARIABLE_#=0;
            CURRENT_ROW#,CURRENT_COL#=0;
            EXPRESSION_AREA = EMPTY ;
            GO TO RETURN_TO_PARSER ;

ACTION_10:  CURRENT_OP = PLUS;
            GO TO RETURN_TO_PARSER;

ACTION_11:  CURRENT_OP = MINUS;
            GO TO RETURN_TO_PARSER;

ACTION_12:  /* THIS ROUTINE BUILDS THE EXPRESSION TREE.
            IT ALLOCATES A NODE FOR THE OPERATOR, INSERTS THE TYPE OF
            OPERATOR FROM THE CURRENT_OP FIELD, DETERMINES THE TYPE
            OF EXPRESSION AND NUMBER OF DIMENSIONS AND PLACES
            THEM IN THE NODE.
            IT LINKS THE PROPER SUBTREE(S) TO THE OPERATOR NODE AND
            PLACES THE RESULT IN THE TOP OF THE PARSING STACK.
            SOME SIMPLE ERROR CHECKING IS ALSO DONE.                  */
            /* SUBTREE_TYPE CODES ARE AS FOLLOWS:
                    SCALAR          0
                    FUNCTION        1
                    COL VECTOR      2
                    ROW VECTOR      3
                    MATRIX          4
                    NULL OPERAND    5                    */

            ALLOCATE TREE_NODE IN (EXPRESSION_AREA) ;
            $TYPE_CODE=CURRENT_OP;
            STRING_POINTER , SEQ_#_PTR = NULL ;
            RLINK,LLINK=NULL;
            NEGATE_TAG , TRANSPOSE_TAG , IDENTITY_TAG = NO ;
            #TEMP_TO_FREE,#CEND_STATEMENTS,#REND_STATEMENTS=0;
            CALL GOTO( ACTION_12_ROUTINE(CURRENT_OP) ) ;

            ACTION_12_ROUTINE_PLUS: ACTION_12_ROUTINE_MINUS:
                IF SUBTREE_TYPE(STK_PTR)¬=SUBTREE_TYPE(STK_PTR-1)
                 THEN CALL #ERROR(12);
                XPTR1=SUBTREE_PTR(STK_PTR);
                XPTR2=SUBTREE_PTR(STK_PTR-1);
                IF XPTR1->$#DIMENSIONS¬=XPTR2->$#DIMENSIONS THEN
                 CALL #ERROR(12);
                $#DIMENSIONS=XPTR1->$#DIMENSIONS;
                TYPE_EXP=XPTR1->TYPE_EXP;
                GO TO LINK_BINARY_OP;

            ACTION_12_ROUTINE_DIVIDE:
                IF SUBTREE_TYPE(STK_PTR)¬=0 THEN CALL #ERROR(13);
                XPTR1=SUBTREE_PTR(STK_PTR-1);
                $#DIMENSIONS=XPTR1->$#DIMENSIONS;
                TYPE_EXP=XPTR1->TYPE_EXP;
                GO TO LINK_BINARY_OP;

            ACTION_12_ROUTINE_INNER_PRODUCT:
                IF SUBTREE_TYPE(STK_PTR) ¬=3 &
                 SUBTREE_TYPE(STK_PTR) ¬= 2 |
```

```
      SUBTREE_TYPE(STK_PTR-1) ¬= 3 &
      SUBTREE_TYPE(STK_PTR-1)  ¬= 2 THEN CALL #ERROR(14) ;
     SUBTREE_TYPE(STK_PTR-1)=0;
     $#DIMENSIONS=0;
     TYPE_EXP=SCALAR;
     GOTO LINK_BINARY_OP;

ACTION_12_ROUTINE_EXPONENTIATE:
     IF SUBTREE_TYPE(STK_PTR)¬=0 THEN CALL #ERROR(15);
     IF SUBTREE_TYPE(STK_PTR-1)¬=4 THEN CALL #ERROR(15);
     XPTR1=SUBTREE_PTR(STK_PTR-1);
     $#DIMENSIONS=XPTR1->$#DIMENSIONS;
     TYPE_EXP=MATRIX;
     GO TO LINK_BINARY_OP;

ACTION_12_ROUTINE_NORM:
     IF SUBTREE_TYPE(STK_PTR) < 2 THEN
      CALL #ERROR(17) ;
     SUBTREE_TYPE(STK_PTR)=0;
     $#DIMENSIONS=0;
     TYPE_EXP=SCALAR;
     GO TO LINK_UNIARY_OP ;

ACTION_12_ROUTINE_UNIMINUS:
     XPTR1=SUBTREE_PTR(STK_PTR);
     $#DIMENSIONS=XPTR1->$#DIMENSIONS;
     TYPE_EXP=XPTR1->TYPE_EXP;
     GO TO LINK_UNIARY_OP;

ACTION_12_ROUTINE_TRANSPOSE:
     IF SUBTREE_TYPE(STK_PTR)=2
      THEN SUBTREE_TYPE(STK_PTR),TYPE_EXP=3;
     ELSE IF SUBTREE_TYPE(STK_PTR)=3
      THEN     SUBTREE_TYPE(STK_PTR),TYPE_EXP=2;
     /* IGNORE TRANSPOSE OF A SCALAR */
     ELSE IF SUBTREE_TYPE(STK_PTR) <= 1 THEN DO;
         FREE TREE_NODE;
         GOTO RETURN_TO_PARSER;
         END;
     ELSE TYPE_EXP=MATRIX;
     XPTR1=SUBTREE_PTR(STK_PTR);
     $#DIMENSIONS=XPTR1->$#DIMENSIONS;
     GO TO LINK_UNIARY_OP;

ACTION_12_ROUTINE_MULTIPLY:
     /* RIGHT SUBTREE OF TYPE SCALAR */
     IF SUBTREE_TYPE(STK_PTR)<=1 THEN DO;
         XPTR1=SUBTREE_PTR(STK_PTR-1);
         $#DIMENSIONS=XPTR1->$#DIMENSIONS;
         TYPE_EXP=XPTR1->TYPE_EXP;
         GO TO LINK_BINARY_OP;
         END;

     /* LEFT SUBTREE OF TYPE SCALAR */
     IF SUBTREE_TYPE(STK_PTR-1)<=1 THEN DO;
         SUBTREE_TYPE(STK_PTR-1)=SUBTREE_TYPE(STK_PTR);
         XPTR1=SUBTREE_PTR(STK_PTR);
```

```
            ACTION_12_ROUTINE_SEQUENCE:
                XPTR1=SUBTREE_PTR(STK_PTR);
                TREE_NODE=XPTR1->TREE_NODE;
                SUBTREE_PTR(STK_PTR)=NODE_POINTER;
                SEQ_#_PTR=POINTER_TO_STRING(TEMPSTRING,
                    EXPRESSION_AREA);
                GO TO RETURN_TO_PARSER;

            ACTION_12_ROUTINE_PART_OF:
                XPTR1=SUBTREE_PTR(STK_PTR);
                TREE_NODE=XPTR1->TREE_NODE;
                XPTR1,SUBTREE_PTR(STK_PTR)=NODE_POINTER;
                XPTR1->STRING_POINTER=POINTER_TO_STRING(B_STRING,
                                    EXPRESSION_AREA);
                GO TO RETURN_TO_PARSER;

            ACTION_12_ROUTINE_SEPARATOR:
                $#DIMENSIONS=0;
                SUBTREE_TYPE(STK_PTR-1)=0;
                GO TO LINK_BINARY_OP;

            ACTION_12_ROUTINE_FUNCTION:
                IF SUBTREE_TYPE(STK_PTR-1) = 1 THEN DO ;
                    SUBTREE_TYPE(STK_PTR-1) = 0 ;
                    $#DIMENSIONS=0;
                    GO TO LINK_BINARY_OP;
                    END;
                ELSE CALL #ERROR(19) ; /* ILLEGAL FUNCTION */
                GO TO RETURN_TO_PARSER;

            LINK_UNIARY_OP:
                RLINK=SUBTREE_PTR(STK_PTR);
                SUBTREE_PTR(STK_PTR)=NODE_POINTER;
                IF STK_PTR = 0 THEN CALL #ERROR(18) ;
                GO TO RETURN_TO_PARSER;

            LINK_BINARY_OP:
                LLINK=SUBTREE_PTR(STK_PTR-1);
                RLINK=SUBTREE_PTR(STK_PTR);
                STK_PTR=STK_PTR-1;
                IF STK_PTR = 0 THEN CALL #ERROR(18) ;
                SUBTREE_PTR(STK_PTR)=NODE_POINTER;
                GO TO RETURN_TO_PARSER;

            /* INSERT MULTIPLY DELIMITER MARKER                    */
ACTION_13:  SUBTREE_TYPE(STK_PTR+1)=SUBTREE_TYPE(STK_PTR);
            SUBTREE_PTR(STK_PTR+1)=SUBTREE_PTR(STK_PTR);
            SUBTREE_TYPE(STK_PTR)=MARKER;
            STK_PTR=STK_PTR+1;
            GO TO RETURN_TO_PARSER;

ACTION_14:  CURRENT_OP=MULTIPLY;
            GO TO RETURN_TO_PARSER;

ACTION_15:  CURRENT_OP=DIVIDE;
            GO TO RETURN_TO_PARSER ;
```

```
            /* TEST PRECEDENCE TABLE FOR MULTIPLY , OR PROCESS        */
            /* A SCALAR DIVIDE                                        */
ACTION_16:  IF CURRENT_OP = DIVIDE |
             PRECEDENCE_TABLE(SUBTREE_TYPE(STK_PTR-1),
              SUBTREE_TYPE(STK_PTR))=0 THEN DO ;
                IF SUBTREE_TYPE(STK_PTR-1)=3 &
                 SUBTREE_TYPE(STK_PTR) ¬= 2 &
                 SUBTREE_TYPE(STK_PTR-2)=0 THEN DO ;
                STK_PTR=STK_PTR-1;
                CALL ACT(12);
                SUBTREE_TYPE(STK_PTR+1)=SUBTREE_TYPE(STK_PTR+2);
                SUBTREE_PTR(STK_PTR+1)=SUBTREE_PTR(STK_PTR+2);
                STK_PTR=STK_PTR+1;
                END;
            CALL ACT(12) ;
            IF SUBTREE_TYPE(STK_PTR-1)=MARKER THEN GO TO
             RETURN_TO_PARSER;
            ELSE CURRENT_OP=MULTIPLY;
            GO TO ACTION_16;
            END ;
            GO TO RETURN_TO_PARSER ;

            /* UNSTACK ALL REMAINING MULTIPLICANDS DOWN TO MARKER     */
            /* AND REMOVE THE MARKER                                  */
ACTION_17:  IF SUBTREE_TYPE(STK_PTR-1)=MARKER THEN DO;
                SUBTREE_TYPE(STK_PTR-1)=SUBTREE_TYPE(STK_PTR);
                SUBTREE_PTR(STK_PTR-1)=SUBTREE_PTR(STK_PTR);
                STK_PTR=STK_PTR-1;
                GO TO RETURN_TO_PARSER;
                END;
            IF PRECEDENCE_TABLE(SUBTREE_TYPE(STK_PTR-1),
             SUBTREE_TYPE(STK_PTR))¬=2 THEN DO;
                CALL ACT(12);
                GO TO ACTION_17;
                END;
            CALL #ERROR(17);
                GO TO RETURN_TO_PARSER ;

ACTION_18:  CURRENT_OP=EXPONENTIATE;
            GO TO ACTION_12;

ACTION_19:  CURRENT_OP=UNIMINUS;
            GO TO ACTION_12;

            /* CHECK FOR AN INNER_PRODUCT CONSTRUCT STARTING AT INP   */
            /*  , IF FOUND SET OK = 0                                 */
ACTION_20:  PARN_COUNT=1;
            ACT_20_INDEX=INP;
            ACT_20_COMMA_CHK:
                IF CHAR(ACT_20_INDEX)=',' & PARN_COUNT =1 THEN DO;
                    OK=0;
                    GO TO RETURN_TO_PARSER;
                    END;
                ELSE
                IF CHAR(ACT_20_INDEX)='(' THEN PARN_COUNT=PARN_COUNT+1;
            ELSE IF CHAR(ACT_20_INDEX)=')' THEN DO;
                    PARN_COUNT=PARN_COUNT-1;
```

```
                IF PARN_COUNT=0 THEN DO;
                    OK=1;
                    GO TO RETURN_TO_PARSER;
                    END;
                END;
            ACT_20_INDEX=ACT_20_INDEX+1;
            GO TO ACT_20_COMMA_CHK;

ACTION_21:  CURRENT_OP=TRANSPOSE;
            GO TO ACTION_12;

ACTION_22:  CURRENT_OP=SEQUENCE;
            GO TO ACTION_12;

ACTION_23:  /* SEE IF IDENTIFIER IS OL/2 VARIABLE
            IF SO STACK VARIABLE NODE ON PARSING STACK  */
            XPTR1 = SEARCH(TEMPIDENT,J);
            IF XPTR1=NULL THEN DO;
                OK=0;
                GO TO RETURN_TO_PARSER;
                END;
            IF XPTR1 -> $TYPE_CODE = BLOCK_ARRAY | XPTR1 ->
             $TYPE_CODE = VECTOR_SPACE THEN CALL #ERROR(
             NOT_IMPLEMENTED ) ;
            SP = XPTR1 -> STRING_POINTER  ;
            B_STRING = STRINGS  ;
            SUBTREE_PTR(STK_PTR+1)=XPTR1;
            SUBTREE_TYPE(STK_PTR+1)=XPTR1->TYPE_EXP;
            STK_PTR = STK_PTR + 1 ;
            IF ON_RIGHT_HAND_SIDE & XPTR1=LHS_IDENT THEN
                LHS_TEMP_NEEDED=YES;
            IF STK_PTR > MAX_STACK THEN CALL #ERROR(100) ;
            GO TO RETURN_TO_PARSER ;

ACTION_24:  /* SET POINTER TO BEGINNING OF A SCALAR STRING ,          */
            /*  AND STACK A MARKER                                   */
            SCALAR_EXP_POINTER = INP ;
            STK_PTR = STK_PTR + 1 ;
            SUBTREE_TYPE(STK_PTR) = SCALAR_STRING_MARKER ;
            GO TO RETURN_TO_PARSER                                  ;

ACTION_25:  /* TAKE THE SCALAR STRINGS FROM ALL THE NODES ABOVE THE
            MARKER IN THE PARSING STACK AND PLACE THEM IN ONE NODE */
            IF SCALAR_EXP_POINTER ¬= INP THEN CALL
             BUILD_AND_STACK_SCALAR_NODE( SCALAR_EXP_POINTER , INP-1 );
            IF SUBTREE_TYPE(STK_PTR-1)¬=SCALAR_STRING_MARKER THEN DO;
                XPTR1=SUBTREE_PTR(STK_PTR);
                XPTR2=XPTR1->STRING_POINTER;
                B_STRING=XPTR2->STRINGS;
                FREE XPTR1->TREE_NODE  IN (EXPRESSION_AREA);
                DO WHILE (SUBTREE_TYPE(STK_PTR-1)¬=
                 SCALAR_STRING_MARKER);
                   XPTR1= SUBTREE_PTR(STK_PTR-1);
                   XPTR2=XPTR1->STRING_POINTER;
                   A_STRING=XPTR2->STRINGS;
                   FREE XPTR1->TREE_NODE IN (EXPRESSION_AREA);
                   A_STRING=A_STRING||B_STRING;
```

```
                B_STRING=A_STRING;
            STK_PTR=STK_PTR-1;
                END;
                STK_PTR=STK_PTR-1;
            CALL BUILD_AND_STACK_STRING_NODE(B_STRING);
            END;
        STK_PTR = STK_PTR - 1 ;
        SUBTREE_TYPE(STK_PTR) = SUBTREE_TYPE(STK_PTR+1) ;
        SUBTREE_PTR(STK_PTR) = SUBTREE_PTR(STK_PTR+1) ;
        GO TO RETURN_TO_PARSER                                          ;

ACTION_26: /* SAVE A POINTER TO THE BEGINNING OF THE IDENTIFIER      */
        SAVER_POINTER = INP ;
        ELEMENT_EXPRESSION_FOUND = NO ;
        GO TO RETURN_TO_PARSER                                          ;

ACTION_27: /* PROCESS PARENTHESIZED CONSTRUCT AFTER THE IDENTIFIER   */
        IF TYPE_OF_ID ¬= OL2_ID THEN GO TO BNOT                         ;
            IF SCALAR_EXP_POINTER ¬= SAVER_POINTER THEN
             CALL BUILD_AND_STACK_SCALAR_NODE(
             SCALAR_EXP_POINTER , SAVER_POINTER - 1 ) ;
ACTION_70:   A_STRING = ''                                              ;
             PARN_COUNT = 1                                             ;
             DO WHILE ( PARN_COUNT > 0 )                                ;
                  CALL SCAN_UNTIL_KEEP( ')' , '(' , ';' )               ;
                  A_STRING = A_STRING || TEMPSTRING                     ;
                  CHARACTER_SCANNED = CHAR(INP)                         ;
                  IF CHARACTER_SCANNED = ')' THEN DO                    ;
                       PARN_COUNT = PARN_COUNT-1                        ;
                       IF PARN_COUNT = 0 THEN GO TO BREADY              ;
                  END                                                   ;
                  ELSE IF CHARACTER_SCANNED = '(' THEN
                  PARN_COUNT = PARN_COUNT + 1                           ;
                  ELSE IF CHARACTER_SCANNED = ';' THEN CALL
                   #ERROR( UNMATCHED_PARNS )                            ;
             END                                                        ;
             BREADY: IF WHICH_ACTION_NUM = 70 THEN GO TO
              RETURN_TO_PARSER                                          ;
            INP = INP + 1 ;
            SAVER_POINTER = INP  ;
            SCALAR_EXP_POINTER=INP;
            CALL BUILD_AND_STACK_STRING_NODE( '@OLELEM'  ||
             '(' || TEMPIDENT || ',' || A_STRING ||')' );
            ELEMENT_EXPRESSION_FOUND = YES ;
            GO TO SET_OK_ZERO_AND_RETURN ;
        BNOT:
            IF SCALAR_EXP_POINTER ¬= SAVER_POINTER THEN CALL
             BUILD_AND_STACK_SCALAR_NODE( SCALAR_EXP_POINTER ,
             SAVER_POINTER - 1 ) ;
            CALL BUILD_AND_STACK_SCALAR_NODE( SAVER_POINTER ,
             INP - 1 ) ;
            SUBTREE_TYPE ( STK_PTR ) = FUNCTION  ;
        GO TO RETURN_TO_PARSER                                          ;

ACTION_28: /* SEE IF PL1 FUNCTION WITH OL2 ARGUMENT */
        IF SUBTREE_TYPE(STK_PTR)¬=SCALAR & TYPE_OF_ID = OTHER
         THEN CALL #ERROR(221);
```

```
            GO TO RETURN_TO_PARSER                                          ;

ACTION_29:  /* BUILD AN ARGUEMENT SUBTREE                    */
            CURRENT_OP= SEPARATOR ;
            GO TO ACTION_12 ;

ACTION_30:  /* FINISH OFF A FUNCTION SUBTREE                                */
            CURRENT_OP = FUNCTION_TYPE ;
            GO TO TO_34;

ACTION_31:  /* HAS AN OL/2 IDENTIFIER ALONE BEEN FOUND ?                    */
            IF TYPE_OF_ID=OL2_ID & ELEMENT_EXPRESSION_FOUND THEN DO;
                INP = SAVER_POINTER  ;
                GO TO RETURN_TO_PARSER;
            END  ;
            IF TYPE_OF_ID = OL2_ID & ¬ ELEMENT_EXPRESSION_FOUND
             THEN DO ;
                IF SAVE_IDENT -> $#DIMENSIONS = 0 THEN DO  ;
                    IF SCALAR_EXP_POINTER ¬= SAVER_POINTER THEN
                    CALL BUILD_AND_STACK_SCALAR_NODE (
                      SCALAR_EXP_POINTER , SAVER_POINTER - 1 )  ;
                    SP = SAVE_IDENT -> STRING_POINTER  ;
                    A_STRING = STRINGS  ;
                    CALL BUILD_AND_STACK_STRING_NODE (
                      '@OLSCAL(' || A_STRING || ')' )  ;
                    SCALAR_EXP_POINTER = INP  ;
                    OK = 1  ;
                END  ;
                ELSE  DO  ;
                    OK = 0  ;
                    FREE TYPE_OF_ID  ;
                END  ;
            END ;
            GO TO RETURN_TO_PARSER                                          ;

ACTION_32:  /* INITIALIZE TO TEST FOR AN ARITHMETIC CONSTANT                */
            TEMPCONST = '' ;
            GO TO RETURN_TO_PARSER                                          ;

ACTION_33:  /* HAS ONLY A '.' BEEN FOUND ?                                  */
            IF TEMPCONST = '' THEN OK = 0 ;
            GO TO RETURN_TO_PARSER                                          ;

ACTION_34:  /* PROCESS A NORM                                               */
            CURRENT_OP = NORM ;
            /* INSERT AN ASSIGNMENT OF NORM, INNER-PRODUCT, OR
            SCALAR FUNCTION TO A TEMPORARY SCALAR VARIABLE.
            CALL COMPILER AND THEN PLACE THE NAME OF THE
            TEMPORARY SCALAR ON THE PARSING STACK */
      TO_34: CALL ACT(12);
            SCALAR_EXP_POINTER=INP;
            SUBTREE_TYPE(STK_PTR+1)=SUBTREE_TYPE(STK_PTR);
            SUBTREE_PTR(STK_PTR+1)=SUBTREE_PTR(STK_PTR);
            STK_PTR=STK_PTR-1;
            CALL BUILD_AND_STACK_STRING_NODE (
            '#TEMPO' || DIGIT_STRINGS(SCALAR_TEMP_VARIABLE_#));
            STK_PTR=STK_PTR+1;
```

```
            CURRENT_OP=EQUAL;
                CALL ACT(12);
            CALL CODER(STK_PTR,SUBTREE_PTR(STK_PTR),
                VECTOR_TEMP_VARIABLE_#,SCALAR_TEMP_VARIABLE_#,
                MATRIX_TEMP_VARIABLE_#,CURRENT_COL#,CURRENT_ROW#);
            CALL BUILD_AND_STACK_STRING_NODE(
            '#TEMPO' || DIGIT_STRINGS(SCALAR_TEMP_VARIABLE_#));
            SCALAR_TEMP_VARIABLE_#=SCALAR_TEMP_VARIABLE_#+1;
            GO TO RETURN_TO_PARSER;

ACTION_35: /* MOVE OVER A ',' OR A ')'                                  */
            SCALAR_EXP_POINTER = INP+1 ;
            GO TO RETURN_TO_PARSER;

 ACTION_36: CURRENT_OP=INNER_PRODUCT;
            GO TO TO_34;

ACTION_37: /* PROCESS A POSSIBLE PREVIOUS SCALAR STRING                 */
            K = INP - 3 ;
TO_37:      IF SCALAR_EXP_POINTER ¬= INP THEN DO ;
                CALL BUILD_AND_STACK_SCALAR_NODE(
                 SCALAR_EXP_POINTER , K );
            END ;
            GO TO RETURN_TO_PARSER                                    ;

ACTION_38: /* DON'T NEED THIS ANY MORE                                 */
            FREE TYPE_OF_ID ;
            GO TO RETURN_TO_PARSER                                    ;

ACTION_39: /* FIND THE TYPE OF THE IDENTIFIER                          */
            ALLOCATE TYPE_OF_ID ;
            TEMP_POINTER1 = SEARCH( TEMPIDENT , I ) ;
            IF TEMP_POINTER1 = NULL THEN DO ;
                IF IS_AN_OL2_ENTRY( TEMPIDENT ) THEN
                 TYPE_OF_ID = OL2_ENTRY ;
                ELSE TYPE_OF_ID = OTHER ;
            END ;
            ELSE DO  ;
                TYPE_OF_ID = OL2_ID  ;
                SAVE_IDENT = TEMP_POINTER1  ;
            END  ;
            GO TO RETURN_TO_PARSER                                    ;

ACTION_40: /* UNSTACK SCALAR MARKER                                    */
            STK_PTR = STK_PTR - 1 ;
            GO TO SET_OK_ZERO_AND_RETURN ;

ACTION_41: /* SAVE PTR TO LEFT HAND SIDE RESULT */
            LHS_IDENT=SUBTREE_PTR(STK_PTR);
            ON_RIGHT_HAND_SIDE=YES;
            GO TO RETURN_TO_PARSER;

ACTION_42:
            /* INSERT ASSIGNMENT TO TEMPORARY VARIABLE WHEN SAME
            VARIABLE WAS USED ON BOTH SIDES OF = SIGN  */
            IF LHS_TEMP_NEEDED THEN DO;
                ALLOCATE TREE_NODE IN (EXPRESSION_AREA);
```

```
            XPTR1=SUBTREE_PTR(STK_PTR-1);
            TREE_NODE=XPTR1->TREE_NODE;
            IF TYPE_EXP<4 THEN DO;
                STRING_POINTER=POINTER_TO_STRING('$TEMP1' ||
                DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),
                 EXPRESSION_AREA);
                VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
                END;
            ELSE DO;
                STRING_POINTER=POINTER_TO_STRING('$TEMP2' ||
                 DIGIT_STRINGS(MATRIX_TEMP_VARIABLE_#),
                 EXPRESSION_AREA);
                MATRIX_TEMP_VARIABLE_#=MATRIX_TEMP_VARIABLE_#+1;
                END;
            SUBTREE_PTR(STK_PTR+1)=SUBTREE_PTR(STK_PTR);
            SUBTREE_TYPE(STK_PTR+1)=SUBTREE_TYPE(STK_PTR);
            SUBTREE_PTR(STK_PTR)=NODE_POINTER;
            SUBTREE_TYPE(STK_PTR)=TYPE_EXP;
            STK_PTR=STK_PTR+1;
            END;
        /* HANDLE ASSIGNMENT, MULTIPLR IF NECESSARY */
        DO I = STK_PTR TO 2 BY -1 ;
            CURRENT_OP = EQUAL ;
            CALL ACT(12) ;
        END ;
        /* CALL COMPILER */
        CALL CODER(STK_PTR,SUBTREE_PTR(STK_PTR),
            VECTOR_TEMP_VARIABLE_#,SCALAR_TEMP_VARIABLE_#,
            MATRIX_TEMP_VARIABLE_#,CURRENT_COL#,CURRENT_ROW#);
        PUT SKIP(2);
        IF VECTOR_TEMP_VARIABLE_#-1>MAX_VECTOR_TEMP THEN
         MAX_VECTOR_TEMP=VECTOR_TEMP_VARIABLE_#-1;
        IF MATRIX_TEMP_VARIABLE_#-1>MAX_MATRIX_TEMP THEN
         MAX_MATRIX_TEMP=MATRIX_TEMP_VARIABLE_#-1;
        ON_RIGHT_HAND_SIDE=NO;
        LHS_TEMP_NEEDED=NO;
        GO TO RETURN_TO_PARSER ;

ACTION_44: /* PROCESS PART-OF CONSTRUCT                              */
        CURRENT_OP = PART_OF ;
        GO TO ACTION_12 ;

ACTION_46: /* PROCESS A "NULL" OPERAND                               */
        STK_PTR = STK_PTR +1 ;
        SUBTREE_PTR(STK_PTR) = $OL2NULL ;
        SUBTREE_TYPE(STK_PTR) = 5 ;
        GO TO RETURN_TO_PARSER ;

ACTION_48: /* INSURE THAT THAT WHICH FOLLOWS IS AN INNER PRODUCT */
        I=INP;
    ACT48_CHK:
        IF CHAR(I)=',' & PARN_COUNT=1 THEN DO;
            OK=1;
            K=INP-2;
            GOTO TO_37;
            END;
        ELSE IF CHAR(I)='(' THEN PARN_COUNT=PARN_COUNT+1;
```

```
        $#DIMENSIONS=XPTR1->$#DIMENSIONS;
        TYPE_EXP=XPTR1->TYPE_EXP;
        GO TO LINK_BINARY_OP;
        END;

    /* LEFT SUBTREE OF TYPE COLUMN VECTOR */
    IF SUBTREE_TYPE(STK_PTR-1)=2 THEN DO;
        IF SUBTREE_TYPE(STK_PTR)=3 THEN DO;
            SUBTREE_TYPE(STK_PTR-1)=4;
            $#DIMENSIONS=2;
        TYPE_EXP=MATRIX;
            GO TO LINK_BINARY_OP;
            END;
        ELSE CALL #ERROR(16);
        GO TO RETURN_TO_PARSER;
        END;

    /* LEFT SUBTREE OF TYPE ROW VECTOR */
    IF SUBTREE_TYPE(STK_PTR-1)=3 THEN DO;
        IF SUBTREE_TYPE(STK_PTR)=2 THEN GO TO ACTION_36;
        IF SUBTREE_TYPE(STK_PTR)=4 THEN DO;
            XPTR1=SUBTREE_PTR(STK_PTR);
            $#DIMENSIONS=XPTR1->$#DIMENSIONS-1;
        IF $#DIMENSIONS=1 THEN DO;
             SUBTREE_TYPE(STK_PTR-1)=3 ;
            TYPE_EXP=ROW_VEC ;  END;
        ELSE SUBTREE_TYPE(STK_PTR-1),TYPE_EXP=MATRIX;
            GO TO LINK_BINARY_OP;
            END;
        END;

    /* LEFT SUBTREE OF TYPE MATRIX */
    IF SUBTREE_TYPE(STK_PTR-1)=4 THEN DO;
        TYPE_EXP=MATRIX;
        XPTR1=SUBTREE_PTR(STK_PTR-1);
        XPTR2=SUBTREE_PTR(STK_PTR);
        $#DIMENSIONS=XPTR1->$#DIMENSIONS+
         XPTR2->$#DIMENSIONS-2;
        IF $#DIMENSIONS=1 THEN
         SUBTREE_TYPE(STK_PTR-1),TYPE_EXP=COL_VEC;
        GO TO LINK_BINARY_OP;
        END;

ACTION_12_ROUTINE_EQUAL:
    IF SUBTREE_TYPE(STK_PTR) = SUBTREE_TYPE(STK_PTR-1) |
     SUBTREE_TYPE(STK_PTR) = 5 & SUBTREE_TYPE(STK_PTR-1)
     > 1 | SUBTREE_TYPE(STK_PTR) = 0 & SUBTREE_TYPE
     (STK_PTR-1) = 4 THEN DO ;
    XPTR1 = SUBTREE_PTR(STK_PTR-1) ;
    IF XPTR1 -> IDENTITY_TAG THEN CALL #ERROR(21) ;
    $#DIMENSIONS=XPTR1->$#DIMENSIONS;
    TYPE_EXP=XPTR1->TYPE_EXP;
    GO TO LINK_BINARY_OP;
    END ;
    CALL #ERROR(20) ;
    GO TO RETURN_TO_PARSER;
```

```
            ELSE IF CHAR(I)=')' THEN DO;
                PARN_COUNT=PARN_COUNT-1;
                IF PARN_COUNT=0 THEN DO;
                    OK=0;  GOTO RETURN_TO_PARSER;  END;  END;
            I=I+1;
            IF CHAR(I)=';' THEN CALL #ERROR(UNMATCHED_PARNS);
            GO TO ACT48_CHK;

ACTION_104:/* PICK UP SEQUENCE EXPRESSION                              */
            IF CHAR(INP) = '|' THEN GO TO SET_OK_ZERO_AND_RETURN  ;
            CALL SCAN_UNTIL_KEEP( '|' , ';' )                         ;
            IF CHAR(INP) = ';' THEN CALL #ERROR (211)  ;
            GO TO RETURN_TO_PARSER ;

ACTION_107: /* SKIP TO NEXT ; AND OUTPUT STATEMENT WITH WARNING */
            OUTPUT_BUFFER=
              '/* **STATEMENT NOT RECOGNIZED AS PL1 OR OL2** */';
            CALL SKIP_AND_OUTPUT;
            K=INP;
            CALL SCAN_UNTIL_PASS( ';' )  ;
            IF CHAR(INP) = ' ' THEN GO TO SET_OK_ZERO_AND_RETURN  ;
            CALL MOVCHAR(OUTPUT_BUFFER,K,INP)  ;
            L107: CALL SKIP_AND_OUTPUT ;
            INP = INP + 1 ;
            GO TO RETURN_TO_PARSER ;

ACTION_113:/*   PROCESS SUBARRAY INDICES                               */
            IF IDENT_DEFINED = NO THEN DO  ;
                CALL SCAN_UNTIL_PASS ( '>' , ';' )  ;
                GO TO RETURN_TO_PARSER  ;
            END  ;
            A_STRING = ''   ;
            PARN_COUNT , COMMA_COUNT = 0   ;
            DO WHILE ( PARN_COUNT >= 0 )  ;
                CALL SCAN_UNTIL_KEEP ( ')' , '(' , ',' , ';' , '>' );
                A_STRING = A_STRING || TEMPSTRING  ;
                IF CHAR(INP) = ')' THEN DO  ;
                    IF PARN_COUNT = 0 THEN DO ;
                        CALL #ERROR (210)  ;
                        CALL SCAN_UNTIL_PASS ( '>' , ';' ) ;
                        PARN_COUNT = -1  ;
                        GO TO NOT_PAST  ;
                    END  ;
                    ELSE PARN_COUNT = PARN_COUNT - 1  ;
                END   ;
                ELSE IF CHAR(INP) = '(' THEN PARN_COUNT =
                 PARN_COUNT + 1   ;
                ELSE IF CHAR(INP) = ',' THEN IF PARN_COUNT = 0 THEN
                 COMMA_COUNT = COMMA_COUNT + 1  ;
                 ELSE ;
                ELSE IF CHAR(INP) ¬= ',' THEN DO  ;
                    IF CHAR(INP) = ';' THEN DO  ;
                        CALL #ERROR (202)  ;
                        IDENT_DEFINED = NO  ;
                        GO TO RETURN_TO_PARSER  ;
                    END  ;
                    PARN_COUNT = -1  ;
```

```
                GO TO NOT_PAST  ;
            END  ;
            A_STRING = A_STRING || CHAR (INP)  ;
            INP = INP + 1  ;
            NOT_PAST: END  ;
        IF COMMA_COUNT = 0 THEN A_STRING = A_STRING || ',0'  ;
        ELSE IF COMMA_COUNT > 1 THEN DO ;
            CALL #ERROR(204)  ;
            IDENT_DEFINED = NO  ;
            GO TO RETURN_TO_PARSER  ;
        END   ;
        B_STRING      = '@OLLOCN (' || B_STRING      || ',' ||
         A_STRING || ')'  ;
        UNQUALIFIED = NO  ;
        GO TO RETURN_TO_PARSER  ;

ACTION_140: /* SCAN OVER PL1 STATEMENTS                              */
        DO WHILE(CHAR(INP)=' ');
            INP=INP+1;      END;
        INP=INP-1;
        K=INP;
         CALL SCAN_UNTIL_PASS(';','%')  ;
        CALL MOVCHAR(OUTPUT_BUFFER,K,INP-1);
        IF CHAR ( INP ) = ';' THEN
                    OUTPUT_BUFFER = OUTPUT_BUFFER || ';'  ;
        CALL SKIP_AND_OUTPUT;
        IF CHAR(INP) = ' ' THEN GO TO RETURN_TO_PARSER;
        INP=INP+1;
        IF CHAR(INP-1)='%' THEN GO TO RETURN_TO_PARSER;
        ELSE GO TO ACTION_140;

SET_OK_ZERO_AND_RETURN: OK = 0                                       ;

RETURN_TO_PARSER: RETURN                                             ;

SKIP_AND_OUTPUT: PROCEDURE                                           ;
     /* SKIP_AND_OUTPUT IS A PROCEDURE WHICH OUTPUTS DATA TO BE
        PASSED TO THE PL/1 COMPILER.  IT WILL NOT BE REPRODUCED HERE*/
END SKIP_AND_OUTPUT                                                  ;

#ERROR: PROCEDURE ( ERROR_CODE_# )                                   ;
     /* #ERROR IS A PROCEDURE TO OUTPUT MESSAGES FOR ERRORS FOUND
        DURING COMPILATION.  IT WILL NOT BE REPRODUCED HERE */
END #ERROR                                                           ;

/* POINTER_TO_STRING ALLOCATES A VARIABLE LENGTH STRING IN A GIVEN
   AREA AND RETURNS A POINTER TO THE STRING  */
POINTER_TO_STRING: PROCEDURE ( STRING , AREA ) RETURNS ( POINTER )  ;
        DECLARE STRING CHARACTER (200) VARYING ,
                AREA AREA(*);
        STRING_LENGTH = LENGTH ( STRING )                            ;
        ALLOCATE VARIABLE_STRING IN ( AREA )                         ;
        STRINGS = STRING                                             ;
        RETURN ( SP )                                                ;
END POINTER_TO_STRING                                                ;

/* SEARCH LOOKS FOR AN IDENTIFIER IN THE OL/2 IDENTIFIER TABLE AND
```

```
   IF IT FINDS IT A POINTER TO THE COMPILE TIME NODE FOR THAT
   VARIABLE IS RETURNED,  ELSE A NULL POINTER IS RETURNED  */
SEARCH: PROCEDURE ( IDENTIFIER , J ) RETURNS ( POINTER )                ;
          DECLARE IDENTIFIER CHAR(32) VARYING , TEMP_PP POINTER ,
           ( J , I ) FIXED BINARY (31,0)                                 ;
          DO I = CURRENT_ID - 1 TO 1 BY -1                               ;
               TEMP_PP = IDENTIFIER_NAME_POINTER(I)                      ;
               IF TEMP_PP -> STRINGS = IDENTIFIER THEN DO                ;
                    J = I                                                ;
                    RETURN ( IDENTIFIER_NODE_POINTER(I) )                ;
               END                                                       ;
          END                                                            ;
          J = 0                                                          ;
          RETURN ( NULL )                                                ;
END SEARCH                                                               ;

/* BUILD_AND_STACK PLACES A NODE ON THE PARSING STACK FOR A
   STRING GIVEN IN THE PARAMETERS  */
BUILD_AND_STACK_SCALAR_NODE: PROCEDURE ( INP1 , INP2 )                   ;
   DCL ( INP1 , INP2 ) FIXED BINARY (31,0)                               ;
          A_STRING = ''                                                  ;
          CALL MOVCHAR( A_STRING , INP1 , INP2 )                         ;
          GO TO LAB1                                                     ;
BUILD_AND_STACK_STRING_NODE: ENTRY ( STRING )                            ;
   DCL STRING CHAR (100) VARYING                                         ;
          A_STRING = STRING                                              ;
          LAB1: STK_PTR = STK_PTR + 1                                    ;
          IF STK_PTR > MAX_STACK THEN CALL #ERROR(100)                   ;
          ALLOCATE TREE_NODE IN ( EXPRESSION_AREA )                      ;
          SUBTREE_PTR( STK_PTR ) = NODE_POINTER                          ;
          SUBTREE_TYPE( STK_PTR ) = SCALAR                               ;
          LLINK , RLINK = NULL                                           ;
          $#DIMENSIONS = 0                                               ;
          STRING_POINTER = POINTER_TO_STRING( A_STRING ,
           EXPRESSION_AREA )                                             ;
          #_OF_TIMES_TO_USE = UNDEFINED                                  ;
          $TYPE_CODE,TYPE_EXP=SCALAR;
          SEQ_#_PTR = NULL                                               ;
          TRANSPOSE_TAG , NEGATE_TAG , IDENTITY_TAG = NO                 ;
END BUILD_AND_STACK_SCALAR_NODE                                          ;

/* THIS PROCEDURE DETERMINES IF A FUNCTION IS TYPE OL/2 */
IS_AN_OL2_ENTRY: PROCEDURE ( TEMPIDENT ) RETURNS ( BIT (1) )             ;
   DCL TEMPIDENT CHAR(32) VARYING                                        ;
          DCL OL2_ENTRIES(5) CHAR(7) INITIAL ('MAXRSUB', 'MAXCSUB',
              'MINRSUB', 'MINCSUB', ' ');
          DO I = 1 TO 5;
              IF OL2_ENTRIES(I)=TEMPIDENT THEN RETURN('1'B);
              END;
          RETURN ('0'B )                                                 ;
END IS_AN_OL2_ENTRY                                                      ;
```

APPENDIX C

OL/2 CODE GENERATOR

```
CODER: PROCEDURE (STK_PTR,TREE_PTR,VECTOR_TEMP_VARIABLE_#,
                  SCALAR_TEMP_VARIABLE_#, MATRIX_TEMP_VARIABLE_#
                  ,CURRENT_COL#, CURRENT_ROW#);
      DCL (A_STRING, B_STRING, C_STRING) CHAR(100) VARYING,
         EXPRESSION_AREA AREA(1500),
         OUTPUT_BUFFER CHAR(118) VARYING,
         (TREE_PTR, XPTR1, XPTR2) POINTER,
         HOLD2 POINTER,
         (SCALAR INITIAL(0), COL_VEC INITIAL(2), ROW_VEC
         INITIAL(3), MATRIX INITIAL(4), CURRENT_ROW#,
         STRING_LENGTH, LOOP_DEPTH,  #LINES,
         CURRENT_COL#, VECTOR_TEMP_VARIABLE_#,
         MATRIX_TEMP_VARIABLE_#, SCALAR_TEMP_VARIABLE_#)
         FIXED BIN (15,0),
         STK_PTR FIXED BIN (31,0),
        (LSEQ,RSEQ,RSLTSEQ) CHAR(100) VARYING,
         DIGIT_STRINGS (0:25) CHAR(2) VARYING INITIAL
        ('0' , '1' , '2' ,
        '3' , '4' , '5' , '6' , '7' , '8' , '9' , '10' , '11' ,
        '12' , '13' , '14' , '15' , '16' , '17' , '18' , '19' ,
        '20' , '21' , '22' , '23' , '24' , '25' )            ,
       1 TREE_NODE BASED (CURRENT_NODE),
         ( 2 RLINK  , 2 LLINK  , 2 SEQ_#_PTR,
           2 STRING_POINTER ) POINTER,
         ( 2 $#DIMENSIONS,
          2 PART_SIZE,
           2 #_OF_TIMES_TO_USE  ,
           2 TYPE_EXP ,
           2 #TEMP_TO_FREE,   2 #CEND_STATEMENTS,
           2 #REND_STATEMENTS,
           2 $TYPE_CODE ) FIXED BINARY (15,0) ,
         ( 2 NEGATE_TAG , 2 TRANSPOSE_TAG,
           2 IDENTITY_TAG ) BIT (1)    ,
       1 VARIABLE_STRING BASED (SP),
           2 LEN FIXED BIN (15,0),
           2 STRINGS CHAR (STRING_LENGTH REFER (LEN))         ,
           (BEEN_THROUGH_HERE_BEFORE INITIAL ('0'B),
           YES INITIAL('1'B)) BIT(1) ALIGNED STATIC;
        DCL TRACEON BIT(1) ALIGNED EXTERNAL;
        DCL POINTER_TOCSTRING ENTRY (CHAR(100) VARYING,)
             RETURNS (POINTER);
        DCL CHAR1 CHAR(1), CHAR2 CHAR(2);
        DCL T_STRING CHAR(5);
        DCL CODER_OPER(-14:-1,0:1,0:1) LABEL STATIC;
        DECLARE (ROOT,CURRENT_NODE) POINTER,
            (TYPE_RIGHT,TYPE_LEFT,OPERATOR INITIAL (0),
            VARIABLE INITIAL (1) ) FIXED BINARY (15,0);
        DECLARE TRAVERSE_STK POINTER CONTROLLED;
        DCL 1 RESULT_STACK CONTROLLED,
            2 PTR_TO_RESULT_NODE POINTER,
```

```
    2 RESULT_SEQ POINTER,
    2 RESULT_TYPE FIXED BINARY (15,0);
DCL HOLD_PTR POINTER, HOLD_TYPE FIXED BIN(15,0);
DCL OUTLINES (20) CHAR(118);
DCL (MULTOO(1:4,1:4),MULTOV(1:4,0:4),MULTVO(0:4,1:4))
 LABEL ;
DCL (LNODE,RNODE) POINTER;
DCL 1 FREE_ARRAY (-1:15),
    (2 LTRANSPOSE, 2 RTRANSPOSE, 2 LNEGATE, 2 RNEGATE)
     CHAR(1),
      2 ASSIGNED FIXED BIN(15,0),
      2 #FREE FIXED BIN (15,0),
      2 FREE_PTR(10) POINTER;
DCL LEVEL# FIXED BIN (15,0);


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 INITIALIZE STATIC LABEL ARRAY TO BE USED AS SWITCH. THIS
 IS DONE ONLY ONCE DURING COMPILATION OF USER'S PROGRAM.
 1ST SUBSCRIPT IS THE TYPE OF OPERATOR
 2ND SUBSCRIPT IS TYPE OF NODE ON LEFT SUBTREE
 3RD SUBSCRIPT IS TYPE OF NODE ON RIGHT SUBTREE
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
IF BEEN_THROUGH_HERE_BEFORE THEN GO TO AROUND_LABEL;
/*  FUNCTION OPERATOR */
CODER_OPER(-14,0,0)=CODER_ILLEGAL;
CODER_OPER(-14,0,1)=CODER_ILLEGAL;
CODER_OPER(-14,1,0)=CODER_FUNC_VAR_OP;
CODER_OPER(-14,1,1)=CODER_FUNC_VAR_VAR;
/*  FUNCTION ARGUMENT SEPARATOR OPERATOR */
CODER_OPER(-13,0,0)=CODER_SEP_OP_OP;
CODER_OPER(-13,0,1)=CODER_SEP_OP_VAR;
CODER_OPER(-13,1,0)=CODER_SEP_VAR_OP;
CODER_OPER(-13,1,1)=CODER_SEP_VAR_VAR;
/* NOT USED */
CODER_OPER(-12,0,0)=CODER_ILLEGAL;
CODER_OPER(-12,0,1)=CODER_ILLEGAL;
CODER_OPER(-12,1,0)=CODER_ILLEGAL;
CODER_OPER(-12,1,1)=CODER_ILLEGAL;
CODER_OPER(-11,0,0)=CODER_ILLEGAL;
CODER_OPER(-11,0,1)=CODER_ILLEGAL;
CODER_OPER(-11,1,0)=CODER_ILLEGAL;
CODER_OPER(-11,1,1)=CODER_ILLEGAL;
/*  TRANSPOSE OPERATOR */
CODER_OPER(-10,0,0)=CODER_TRAN_OP;
CODER_OPER(-10,0,1)=CODER_TRAN_VAR;
CODER_OPER(-10,1,0)=CODER_ILLEGAL;
CODER_OPER(-10,1,1)=CODER_ILLEGAL;
/*  UNIARY MINUS OPERATOR */
CODER_OPER(-09,0,0)=CODER_UNIM_OP;
CODER_OPER(-09,0,1)=CODER_UNIM_VAR;
CODER_OPER(-09,1,0)=CODER_ILLEGAL;
CODER_OPER(-09,1,1)=CODER_ILLEGAL;
/*  NORM OPERATOR */
CODER_OPER(-08,0,0)=COL_WALK_RIGHT;
CODER_OPER(-08,0,1)=CODER_NORM_VAR;
CODER_OPER(-08,1,0)=CODER_ILLEGAL;
```

```
      CODER_OPER(-08,1,1)=CODER_ILLEGAL;
      /*  EQUAL OPERATOR */
      CODER_OPER(-07,0,0)=CODER_ILLEGAL;
      CODER_OPER(-07,0,1)=CODER_ILLEGAL;
      CODER_OPER(-07,1,0)=CODER_EQUL_VAR_OP;
      CODER_OPER(-07,1,1)=CODER_EQUL_VAR_VAR;
      /*  INNERPRODUCT OPERATOR */
      CODER_OPER(-06,0,0)=COL_WALK_LEFT;
      CODER_OPER(-06,0,1)=COL_WALK_LEFT;
      CODER_OPER(-06,1,0)=COL_WALK_RIGHT;
      CODER_OPER(-06,1,1)=CODER_IPRD_VAR_VAR;
      /* EXPONENTIATION OPERATOR */
      CODER_OPER(-05,0,0)=CODER_ILLEGAL;
      CODER_OPER(-05,0,1)=CODER_EXPN_OP;
      CODER_OPER(-05,1,0)=CODER_ILLEGAL;
      CODER_OPER(-05,1,1)=CODER_EXPN_VAR;
      /*  MULTIPLY OPERATOR */
      CODER_OPER(-04,0,0)=CODER_MULT_OP_OP;
      CODER_OPER(-04,0,1)=CODER_MULT_OP_VAR;
      CODER_OPER(-04,1,0)=CODER_MULT_VAR_OP;
      CODER_OPER(-04,1,1)=CODER_MULT_VAR_VAR;
      /*  DIVIDE OPERATOR */
      CODER_OPER(-03,0,0)=CODER_ILLEGAL;
      CODER_OPER(-03,0,1)=CODER_DIVD_OP;
      CODER_OPER(-03,1,0)=CODER_ILLEGAL;
      CODER_OPER(-03,1,1)=CODER_DIVD_VAR;
      /*  SUBTRACTION OPERATOR */
      CODER_OPER(-02,0,0)=CODER_SUBT_OP_OP;
      CODER_OPER(-02,0,1)=CODER_SUBT_OP_VAR;
      CODER_OPER(-02,1,0)=CODER_SUBT_VAR_OP;
      CODER_OPER(-02,1,1)=CODER_SUBT_VAR_VAR;
      /*  ADDITION OPERATOR */
      CODER_OPER(-01,0,0)=CODER_ADDS_OP_OP;
      CODER_OPER(-01,0,1)=CODER_ADDS_OP_VAR;
      CODER_OPER(-01,1,0)=CODER_ADDS_VAR_OP;
      CODER_OPER(-01,1,1)=CODER_ADDS_VAR_VAR;
      BEEN_THROUGH_HERE_BEFORE=YES;
      RETURN;

   AROUND_LABEL:
      ROOT,CURRENT_NODE=TREE_PTR;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       IF TREE CONSISTS OF ONLY A VARIABLE THEN RETURN
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF ROOT->$TYPE_CODE >= 0 THEN RETURN;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      INITIALIZE INDICATORS AND COUNTERS
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      DO I = -1 TO 15;
          LTRANSPOSE(I),RTRANSPOSE(I)='0';
          LNEGATE(I),RNEGATE(I)='0';
          ASSIGNED(I)=0;
          #FREE(I)=0;      END;
      LEVEL#,LOOP_DEPTH,#LINES=0;
      ON ERROR BEGIN;
```

```
        ON ERROR CALL IHEDUMP;
         PUT DATA (A_STRING, B_STRING, C_STRING, TYPE_RIGHT,
            TYPE_LEFT, OUTPUT_BUFFER);
         PUT SKIP;
         PUT LIST ('$TYPE_CODE=' ,$TYPE_CODE, 'TYPE_EXP=',
          TYPE_EXP) SKIP;
         DO I = 1 TO #LINES;
             PUT LIST (OUTLINES(I)) SKIP; END;
         CALL CLOSETR;
         END;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       SET TYPES OF RIGHT & LEFT NODE AND BRANCH THROUGH
       SWITCH TO PROPER ROUTINE
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
NEXT_NODE:
      LNODE=LLINK;
      RNODE=RLINK;
      IF LNODE=NULL | LNODE->$TYPE_CODE<0 THEN
       TYPE_LEFT=OPERATOR;
      ELSE TYPE_LEFT=VARIABLE;
      IF RNODE->$TYPE_CODE<0 THEN TYPE_RIGHT=OPERATOR;
      ELSE TYPE_RIGHT=VARIABLE;
      CALL GOTO(CODER_OPER(CURRENT_NODE->$TYPE_CODE,TYPE_LEFT,
        TYPE_RIGHT));


      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       STACK NAME FOR NEW TEMPORARY COL VECTOR AND WALK RIGHT
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
COL_WALK_RIGHT:
MULTVO(4,2): MULTVO(3,2): MULTOO(4,2):
      IF RNODE->$TYPE_CODE=-10 | RNODE->$TYPE_CODE=-9
       THEN GO TO WALK_RIGHT;
      ALLOCATE RESULT_STACK;
      #TEMP_TO_FREE=#TEMP_TO_FREE+1;
      PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP1' ||
       DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),EXPRESSION_AREA);
      RESULT_SEQ=NULL;
      VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
      RESULT_TYPE=COL_VEC;
      GO TO WALK_RIGHT;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       STACK NAME FOR NEW TEMPORARY ROW VECTOR AND WALK RIGHT
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
  ROW_WALK_RIGHT: MULTVO(2,3):
      ALLOCATE RESULT_STACK;
      #TEMP_TO_FREE=#TEMP_TO_FREE+1;
      RESULT_SEQ=NULL;
      PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP1' ||
       DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),EXPRESSION_AREA);
      VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
      RESULT_TYPE=ROW_VEC;
      GO TO WALK_RIGHT;

CODER_ADDS_VAR_OP:
```

```
CODER_SUBT_VAR_OP:
      IF PTR_TO_RESULT_NODE=LNODE->STRING_POINTER THEN DO;
          IF RESULT_TYPE=COL_VEC THEN GO TO COL_WALK_RIGHT;
          IF RESULT_TYPE=ROW_VEC THEN GO TO ROW_WALK_RIGHT;
          END;
MULTVO(0,2): MULTVO(0,3):
CODER_UNIM_OP:
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       IF LEFT SUBTREE IS ROW OR COL OF MATRIX AND RIGHT
       SUBTREE IS MATRIX EXPRESSION GO TO GET NEW TEMPORARY
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF LNODE¬=NULL & LNODE->TYPE_EXP<RESULT_TYPE THEN DO;
          IF LNODE->TYPE_EXP=ROW_VEC THEN GOTO ROW_WALK_RIGHT;
          IF LNODE->TYPE_EXP=COL_VEC THEN GOTO COL_WALK_RIGHT;
          END;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       RESTACK CURRENT TEMPORARY VARIABLE AND WALK RIGHT
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
RESTACK_RIGHT:
      HOLD_PTR=PTR_TO_RESULT_NODE;
      HOLD2=RESULT_SEQ;
      HOLD_TYPE=RESULT_TYPE;
      ALLOCATE RESULT_STACK;
      #TEMP_TO_FREE=#TEMP_TO_FREE+1;
      PTR_TO_RESULT_NODE=HOLD_PTR;
      RESULT_SEQ=HOLD2;
      RESULT_TYPE=HOLD_TYPE;
      GO TO WALK_RIGHT;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       STACK NAME FOR NEW TEMPORARY COL VECTOR AND WALK LEFT
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
COL_WALK_LEFT:
MULTOO(2,3): MULTOV(2,3):
      IF LNODE->$TYPE_CODE=-10 | LNODE->$TYPE_CODE=-9
       THEN GO TO WALK_LEFT;
      ALLOCATE RESULT_STACK;
      #TEMP_TO_FREE=#TEMP_TO_FREE+1;
      PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP1' ||
       DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),EXPRESSION_AREA);
      RESULT_SEQ=NULL;
      VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
      RESULT_TYPE=COL_VEC;
      GO TO WALK_LEFT;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       SET UP ROW PARTITIONING LOOP
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
MULTOV(4,4): MULTOV(4,2):
      IF (RESULT_TYPE=MATRIX | RNODE->$#DIMENSIONS=1) &
       LOOP_DEPTH=0 & LNODE->$TYPE_CODE¬=-10 &
       LNODE->$TYPE_CODE¬=-9 THEN DO;
          CURRENT_ROW#=CURRENT_ROW#+1;
          C_STRING=PTR_TO_RESULT_NODE->STRINGS;
          OUTPUT_BUFFER='DO #ROW' ||DIGIT_STRINGS(
           CURRENT_ROW#) || '=' || C_STRING || '->#LOWER(1) TO '
           || C_STRING || '->#UPPER(1);'   ;
```

```
            CALL SKIP_ANDCOUTPUT;
            LOOP_DEPTH=LOOP_DEPTH+1;
            #REND_STATEMENTS=#REND_STATEMENTS+1;
            END;

        IF RESULT_TYPE=COL_VEC THEN GO TO MATRIX_WALK_LEFT;

        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         STACK NAME FOR NEW TEMPORARY ROW VECTOR AND WALK LEFT
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
ROW_WALK_LEFT:
MULTOO(3,2): MULTOO(3,4): MULTOV(3,4): MULTOV(3,2):
        IF LNODE->$TYPE_CODE=-10 | LNODE->$TYPE_CODE=-9
         THEN GO TO WALK_LEFT;
        ALLOCATE RESULT_STACK;
        #TEMP_TO_FREE=#TEMP_TO_FREE+1;
        PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP1' ||
         DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),EXPRESSION_AREA);
        RESULT_SEQ=NULL;
        VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
        RESULT_TYPE=ROW_VEC;
        GO TO WALK_LEFT;

MULTOV(4,0):
CODER_DIVD_OP:
CODER_ADDS_OP_OP:
CODER_SUBT_OP_OP:
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         SET UP COLUMN PARTITIONING LOOP
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        IF RESULT_TYPE=MATRIX & LNODE->$TYPE_CODE¬=-10 &
          LNODE->$TYPE_CODE¬=-9 THEN DO;
            CURRENT_COL#=CURRENT_COL#+1;
            C_STRING=PTR_TO_RESULT_NODE->STRINGS;
            OUTPUT_BUFFER='DO #COL' || DIGIT_STRINGS(CURRENT_COL#)
            || '=' || C_STRING || '->#LOWER(2) TO ' ||
            C_STRING || '->#UPPER(2);'  ;
            CALL SKIP_ANDCOUTPUT;
            #CEND_STATEMENTS=#CEND_STATEMENTS+1;
            LOOP_DEPTH=LOOP_DEPTH+1;
            GO TO COL_WALK_LEFT;
            END;

CODER_ADDS_OP_VAR:
CODER_SUBT_OP_VAR:
        IF PTR_TO_RESULT_NODE=RNODE->STRING_POINTER THEN DO;
            IF RESULT_TYPE=ROW_VEC THEN GO TO ROW_WALK_LEFT;
            ELSE GO TO COL_WALK_LEFT;   END;
MULTOV(2,0): MULTOV(3,0):
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         RESTACK CURRENT TEMPORARY VARIABLE AND WALK LEFT
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        HOLD_PTR=PTR_TO_RESULT_NODE;
        HOLD2=RESULT_SEQ;
        HOLD_TYPE=RESULT_TYPE;
        ALLOCATE RESULT_STACK;
        #TEMP_TO_FREE=#TEMP_TO_FREE+1;
```

```
      PTR_TO_RESULT_NODE=HOLD_PTR;
      RESULT_SEQ=HOLD2;
      RESULT_TYPE=HOLD_TYPE;
      GO TO WALK_LEFT;

CODER_ADDS_VAR_VAR:
      OUTPUT_BUFFER='CALL @OLADD(' ;
      GO TO SET_OPERANDS;

CODER_SUBT_VAR_VAR:
      OUTPUT_BUFFER='CALL @OLSUB(' ;
      GO TO SET_OPERANDS;

CODER_DIVD_VAR:
      OUTPUT_BUFFER='CALL @OLDIVD(' ;

SET_OPERANDS:
      CALL SET_VARIABLES;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       SET SYSTEM PARTITIONING DATA FOR EACH OPERAND
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
  SET_ROW_OR_COL:
      IF RESULT_TYPE=ROW_VEC THEN DO ;
           C_STRING=C_STRING || '3,0'   ;
          IF LNODE->$#DIMENSIONS=1 THEN
           A_STRING=A_STRING || '3,0'   ;   ELSE
          A_STRING=A_STRING || '3,#ROW' || DIGIT_STRINGS(
           CURRENT_ROW#);
          IF RNODE->$#DIMENSIONS=1 THEN
           B_STRING=B_STRING || '3,0'   ;   ELSE
          B_STRING=B_STRING || '3,#ROW' || DIGIT_STRINGS(
           CURRENT_ROW#);
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       IF BOTH OPERANDS WERE TRUE VECTORS  (NOT PARTITIONED)
       AND ARE INSIDE PARTITIONING LOOP THEN OUTPUT CODE NOW
       SO WON'T BE RECOMPUTED EACH TIME THRU LOOP
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
          IF LNODE->$#DIMENSIONS=1 & RNODE->$#DIMENSIONS=1
              THEN DO;
              IF LOOP_DEPTH>0 THEN DO;
                  IF $TYPE_CODE=-3 THEN DO;
                      B_STRING=XPTR2->STRINGS;
                      B_STRING=B_STRING || ',0,1,0,0,0';  END;
                  OUTPUT_BUFFER=OUTPUT_BUFFER || A_STRING ||
                      ',' || B_STRING || ',' || C_STRING || ');';
                  PUT FILE(SYSPRINT) LIST (OUTPUT_BUFFER) SKIP;
                  OUTPUT_BUFFER='';
                  TYPE_EXP=ROW_VEC;
                  STRING_POINTER=PTR_TO_RESULT_NODE;
                  GO TO WALK_UP;
                  END;
              END;
          CURRENT_NODE->TYPE_EXP=ROW_VEC;   END;
      ELSE IF RESULT_TYPE=COL_VEC THEN DO;
           C_STRING=C_STRING || '2,0'   ;
          IF LNODE->$#DIMENSIONS=1 THEN
           A_STRING=A_STRING || '2,0'   ;   ELSE
```

```
      A_STRING=A_STRING || '2,#COL' || DIGIT_STRINGS(
       CURRENT_COL#)  ;
      IF RNODE->$#DIMENSIONS=1 THEN
       B_STRING=B_STRING || '2,0'   ;     ELSE
      B_STRING=B_STRING || '2,#COL' || DIGIT_STRINGS(
       CURRENT_COL#)  ;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IF BOTH OPERANDS WERE TRUE VECTORS  (NOT PARTITIONED)
 AND ARE INSIDE PARTITIONING LOOP THEN OUTPUT CODE NOW
 SO WON'T BE RECOMPUTED EACH TIME THRU LOOP
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF LNODE->$#DIMENSIONS=1 & RNODE->$#DIMENSIONS=1
          THEN DO;
          IF LOOP_DEPTH>0 THEN DO;
              IF $TYPE_CODE=-3 THEN DO;
                  B_STRING=XPTR2->STRINGS;
                  B_STRING=B_STRING || ',0,1,0,0,0';  END;
              OUTPUT_BUFFER=OUTPUT_BUFFER || A_STRING ||
                  ',' || B_STRING || ',' || C_STRING || ');';
              PUT FILE(SYSPRINT) LIST (OUTPUT_BUFFER) SKIP;
              OUTPUT_BUFFER='';
              TYPE_EXP=COL_VEC;
              STRING_POINTER=PTR_TO_RESULT_NODE;
              GO TO WALK_UP;
              END;
          END;
      CURRENT_NODE->TYPE_EXP=COL_VEC;  END;
  ELSE IF RESULT_TYPE=MATRIX THEN DO;
      IF LNODE->TYPE_EXP=ROW_VEC | RNODE->TYPE_EXP=ROW_VEC
       THEN DO;
          IF LNODE->TYPE_EXP=ROW_VEC THEN A_STRING=A_STRING
              || '3,0';  ELSE
          A_STRING=A_STRING || '3,#ROW' || DIGIT_STRINGS(
              CURRENT_ROW#);
          IF RNODE->TYPE_EXP=ROW_VEC THEN B_STRING=B_STRING
              || '3,0';   ELSE
          B_STRING=B_STRING || '3,#ROW' || DIGIT_STRINGS(
              CURRENT_ROW#);
          C_STRING=C_STRING || '3,#ROW' || DIGIT_STRINGS(
              CURRENT_ROW#);
          END;
      ELSE IF LNODE->TYPE_EXP=COL_VEC | RNODE->TYPE_EXP=
       COL_VEC THEN DO;
          IF LNODE->TYPE_EXP=COL_VEC THEN A_STRING=A_STRING
              || '2,0';  ELSE
          A_STRING=A_STRING || '2,#COL' || DIGIT_STRINGS(
              CURRENT_COL#);
          IF RNODE->TYPE_EXP=COL_VEC THEN B_STRING=B_STRING
              || '2,0';   ELSE
          B_STRING=B_STRING || '2,#COL' || DIGIT_STRINGS(
              CURRENT_COL#);
          C_STRING=C_STRING || '2,#COL' || DIGIT_STRINGS(
              CURRENT_COL#);
          END;
      ELSE DO;
          A_STRING=A_STRING || '4,0';
          B_STRING=B_STRING || '4,0';
```

```
            C_STRING=C_STRING || '4,0';
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         IF BOTH OPERANDS WERE TRUE MATRICES (NOT PARTITIONED)
         AND ARE INSIDE PARTITIONING LOOP THEN OUTPUT CODE NOW
         SO WON'T BE RECOMPUTED EACH TIME THRU LOOP
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
            IF LOOP_DEPTH>0 THEN DO;
                IF $TYPE_CODE=-3 THEN DO;
                    B_STRING=XPTR2->STRINGS;
                    B_STRING=B_STRING || ',0,1,0,0,0';
                    END;
                OUTPUT_BUFFER=OUTPUT_BUFFER || A_STRING ||
                ',' || B_STRING || ',' || C_STRING || ');' ;
                PUT LIST(OUTPUT_BUFFER) SKIP;
                OUTPUT_BUFFER='';
                STRING_POINTER=PTR_TO_RESULT_NODE;
                GO TO WALK_UP;
                END;
            END;
        CURRENT_NODE->TYPE_EXP=MATRIX;   END;
    IF $TYPE_CODE=-3 THEN DO;
        B_STRING=XPTR2->STRINGS;
        B_STRING=B_STRING || ',0,1,0,0,0';
        END;
    OUTPUT_BUFFER=OUTPUT_BUFFER || A_STRING || ',' || B_STRING
     || ',' || C_STRING || ');'    ;
    CALL SKIP_ANDCOUTPUT;
    CURRENT_NODE->STRING_POINTER=PTR_TO_RESULT_NODE;
    GO TO WALK_UP;

CODER_NORM_VAR:
    XPTR2=RNODE->STRING_POINTER;
    B_STRING=XPTR2->STRINGS;
    IF RNODE->SEQ_#_PTR=NULL THEN RSEQ='0';
    ELSE DO;
        SP=RNODE->SEQ_#_PTR;
        RSEQ=STRINGS;   END;
    B_STRING=B_STRING || ',' || RSEQ || ',0,1,' ||
      RNEGATE(LEVEL#) ;
    CURRENT_NODE->STRING_POINTER=POINTER_TOCSTRING(
    '@OLNORM(' || B_STRING || ')' ,EXPRESSION_AREA);
    CURRENT_NODE->TYPE_EXP=SCALAR;
    GO TO WALK_UP;

CODER_IPRD_VAR_VAR:
    CALL SET_VARIABLES;
    CURRENT_NODE->STRING_POINTER=POINTER_TOCSTRING(
     '@OLIPRD(' || A_STRING || ',' || B_STRING ||
     ')' , EXPRESSION_AREA);
    CURRENT_NODE->TYPE_EXP=SCALAR;
    GO TO WALK_UP;

    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     SET NEGATE INDICATORS
     * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CODER_UNIM_VAR:  HOLD_PTR=TRAVERSE_STK;
    IF HOLD_PTR->RLINK=CURRENT_NODE THEN DO;
```

```
          IF RNEGATE(LEVEL#)='0' & LNEGATE(LEVEL#)='0'
           THEN  RNEGATE(LEVEL#-1)='1';
          ELSE RNEGATE(LEVEL#-1)='0'  ;   END;
       ELSE DO; IF LNEGATE(LEVEL#)='0' & RNEGATE(LEVEL#)='0'
          THEN LNEGATE(LEVEL#-1)='1'; ELSE LNEGATE(LEVEL#-1)='0';
           END;
       STRING_POINTER=RNODE->STRING_POINTER;
       GO TO WALK_UP;

CODER_EXPN_OP:
       ALLOCATE RESULT_STACK;
       #TEMP_TO_FREE=#TEMP_TO_FREE+1;
       PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP2' ||
        DIGIT_STRINGS(MATRIX_TEMP_VARIABLE_#),EXPRESSION_AREA);
       RESULT_SEQ=NULL;
       MATRIX_TEMP_VARIABLE_#=MATRIX_TEMP_VARIABLE_#+1;
       RESULT_TYPE=MATRIX;
       GO TO WALK_LEFT;
CODER_EXPN_VAR:
       CALL SET_VARIABLES;
       OUTPUT_BUFFER=OUTPUT_BUFFER || 'CALL @OPEXPN('
       || A_STRING || ',' || B_STRING ||
        ',' || C_STRING || ');'   ;
       CALL SKIP_ANDCOUTPUT;
       TYPE_EXP=MATRIX;
       GO TO WALK_UP;

CODER_TRAN_OP:
       IF RNODE->$TYPE_CODE=-10 THEN GO TO WALK_RIGHT;
       ALLOCATE RESULT_STACK;
       RESULT_SEQ=NULL;
       HOLD_PTR=TRAVERSE_STK;
       HOLD_PTR->#TEMP_TO_FREE=HOLD_PTR->#TEMP_TO_FREE+1;
       IF RNODE->TYPE_EXP=MATRIX THEN DO;
           PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP2' ||
            DIGIT_STRINGS(MATRIX_TEMP_VARIABLE_#),
            EXPRESSION_AREA);
           MATRIX_TEMP_VARIABLE_#=MATRIX_TEMP_VARIABLE_#+1;
           RESULT_TYPE=MATRIX;
           GO TO WALK_RIGHT;
           END;
       ELSE IF RNODE->TYPE_EXP=COL_VEC THEN DO;
           PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP1' ||
           DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),
            EXPRESSION_AREA);
           VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
           RESULT_TYPE=COL_VEC;
           GO TO WALK_RIGHT;
           END;
       ELSE DO;
           PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP1' ||
           DIGIT_STRINGS(VECTOR_TEMP_VARIABLE_#),
            EXPRESSION_AREA);
           VECTOR_TEMP_VARIABLE_#=VECTOR_TEMP_VARIABLE_#+1;
           RESULT_TYPE=ROW_VEC;
           GO TO WALK_RIGHT;   END;
       /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
        SET TRANSPOSE INDICATORS
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CODER_TRAN_VAR: HOLD_PTR=TRAVERSE_STK;
      IF HOLD_PTR->RLINK=CURRENT_NODE THEN DO;
          IF RTRANSPOSE(LEVEL#)='0' & LTRANSPOSE(LEVEL#)='0'
           THEN RTRANSPOSE(LEVEL#-1)='1';
          ELSE RTRANSPOSE(LEVEL#-1)='0';  END;
      ELSE DO;
          IF RTRANSPOSE(LEVEL#)='0' & LTRANSPOSE(LEVEL#)='0'
           THEN LTRANSPOSE(LEVEL#-1)='1';
          ELSE LTRANSPOSE(LEVEL#-1)='0';  END;
      STRING_POINTER=RNODE->STRING_POINTER;
      GO TO WALK_UP;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       SWITCH TO PROPER MULTIPLY ROUTINE BY TYPE OF EXPRESSION
       OR VARIABLE ON THE SUBTREES
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CODER_MULT_OP_OP:
      IF LNODE->TYPE_EXP=SCALAR | RNODE->TYPE_EXP=SCALAR THEN
       GO TO CODER_ILLEGAL;
      GO TO MULTOO(LNODE->TYPE_EXP,RNODE->TYPE_EXP);
CODER_MULT_OP_VAR:
      IF LNODE->TYPE_EXP=SCALAR THEN GO TO CODER_ILLEGAL;
      GO TO MULTOV(LNODE->TYPE_EXP,RNODE->TYPE_EXP);
CODER_MULT_VAR_OP:
      IF RNODE->TYPE_EXP=SCALAR THEN GO TO CODER_ILLEGAL;
      GO TO MULTVO(LNODE->TYPE_EXP,RNODE->TYPE_EXP);

MULTOO(4,4): /* LEFT AND RIGHT OPS MATRICES */
      IF LNODE->$TYPE_CODE=-10 | LNODE->$TYPE_CODE=-9
          THEN GO TO WALK_LEFT;
MATRIX_WALK_LEFT:
      ALLOCATE RESULT_STACK;
      #TEMP_TO_FREE=#TEMP_TO_FREE+1;
      RESULT_SEQ=NULL;
      RESULT_TYPE=MATRIX;
      PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP2' ||
       DIGIT_STRINGS(MATRIX_TEMP_VARIABLE_#),EXPRESSION_AREA);
      MATRIX_TEMP_VARIABLE_#=MATRIX_TEMP_VARIABLE_#+1;
      GO TO WALK_LEFT;
MULTVO(4,4): MULTVO(3,4): MULTVO(0,4):
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       OUTPUT COLUMN PARTITIONING LOOP
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF RESULT_TYPE=MATRIX & RNODE->$TYPE_CODE¬=-10 &
        RNODE->$TYPE_CODE¬=-9 THEN DO;
          CURRENT_COL#=CURRENT_COL#+1;
          C_STRING=PTR_TO_RESULT_NODE->STRINGS;
          OUTPUT_BUFFER='DO #COL' || DIGIT_STRINGS(CURRENT_COL#)
          || '=' || C_STRING || '->#LOWER(2) TO ' ||
          C_STRING || '->#UPPER(2);'  ;
          #CEND_STATEMENTS=#CEND_STATEMENTS+1;
          LOOP_DEPTH=LOOP_DEPTH+1;
          CALL SKIP_ANDCOUTPUT;
          GO TO COL_WALK_RIGHT;   END;
      IF LNODE->TYPE_EXP=SCALAR THEN GO TO RESTACK_RIGHT;
```

```
      ELSE GO TO COL_WALK_RIGHT;

CODER_MULT_VAR_VAR:
      CALL SET_VARIABLES;
      OUTPUT_BUFFER=OUTPUT_BUFFER || 'CALL @OLMULT(' ;

      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       SET UP OPERAND PARTITIONING FOR MULTIPLY
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF LNODE->TYPE_EXP=MATRIX THEN DO;
          /* MATRIX*MATRIX */
          IF RNODE->TYPE_EXP=MATRIX THEN DO;
              IF RESULT_TYPE=COL_VEC THEN DO;
              COL_RESULT:
                  A_STRING=A_STRING || '4,0'  ;
                  B_STRING=B_STRING || '2,#COL' || DIGIT_STRINGS(
                   CURRENT_COL#);
                  C_STRING=C_STRING || '2,0'  ;
                  TYPE_EXP=COL_VEC;
                  GO TO SET_TYPE_WALK_UP;    END;
              IF RESULT_TYPE=ROW_VEC THEN DO;
              ROW_RESULT:
                  A_STRING=A_STRING || '3,#ROW' || DIGIT_STRINGS(
                   CURRENT_ROW#)  ;
                  B_STRING=B_STRING || '4,0'  ;
                  C_STRING=C_STRING || '3,0'  ;
                  TYPE_EXP=ROW_VEC;
                  GO TO SET_TYPE_WALK_UP;    END;
              IF RESULT_TYPE=MATRIX THEN DO;
              MATRIX_RESULT:
                  A_STRING=A_STRING || '4,0'  ;
                  B_STRING=B_STRING || '4,0'  ;
                  C_STRING=C_STRING || '4,0'   ;
                  IF LOOP_DEPTH>0 THEN DO;
                      OUTPUT_BUFFER=OUTPUT_BUFFER || A_STRING
                       || ',' ||B_STRING || ',' || C_STRING ||
                       ');';
                      PUT LIST(OUTPUT_BUFFER) SKIP;
                J=LENGTH(OUTPUT_BUFFER);
               PUT FILE(SYSPUNCH) EDIT (SUBSTR(OUTPUT_BUFFER,
                1,J)) (X(1),A(79));
                      STRING_POINTER=PTR_TO_RESULT_NODE;
                      OUTPUT_BUFFER='';
                      GO TO WALK_UP;
                      END;
                  GO TO SET_TYPE_WALK_UP;    END;
              END;
          /* MATRIX*COLUMN VECTOR */
          IF RNODE->TYPE_EXP=COL_VEC THEN DO;
              A_STRING=A_STRING || '4,0';
              B_STRING=B_STRING || '2,0';
              IF RESULT_TYPE=MATRIX THEN DO;
                  C_STRING=C_STRING || '2,#COL' || DIGIT_STRINGS(
                   CURRENT_COL#);
                  RESULT_TYPE=MATRIX;
                  GO TO SET_TYPE_WALK_UP;
                  END;
```

```
            ELSE DO;
                C_STRING=C_STRING || '2,0';
                END;
            TYPE_EXP=COL_VEC;
            GO TO SET_TYPE_WALK_UP;
            END;
        /* MATRIX*SCALAR */
        IF RNODE->TYPE_EXP=SCALAR THEN DO;
            B_STRING=B_STRING || '0,0';
            IF RESULT_TYPE = MATRIX THEN DO;
                A_STRING=A_STRING || '4,0';
                C_STRING=C_STRING || '4,0';
                GO TO SET_TYPE_WALK_UP;     END;
            IF RESULT_TYPE=COL_VEC THEN DO;
                A_STRING=A_STRING || '2,#COL' || DIGIT_STRINGS(
                 CURRENT_COL#);
                C_STRING=C_STRING || '2,0';
                TYPE_EXP=COL_VEC;
                GO TO SET_TYPE_WALK_UP;     END;
            IF RESULT_TYPE=ROW_VEC THEN DO;
                A_STRING=A_STRING || '3,#ROW' || DIGIT_STRINGS(
                  CURRENT_ROW#);
                C_STRING=C_STRING || '3,0';
                TYPE_EXP=ROW_VEC;
                GO TO SET_TYPE_WALK_UP;     END;
            ELSE GO TO CODER_ILLEGAL;
        END;
        ELSE GO TO CODER_ILLEGAL;
    END;

    IF LNODE->TYPE_EXP=ROW_VEC THEN DO;
        /* ROW VECTOR*MATRIX */
        IF RNODE->TYPE_EXP=MATRIX THEN DO;
            A_STRING=A_STRING || '3,0';
            B_STRING=B_STRING || '4,0';
            IF RESULT_TYPE=MATRIX THEN DO;
                C_STRING=C_STRING || '3,#ROW' || DIGIT_STRINGS(
                 CURRENT_ROW#);
                TYPE_EXP=MATRIX;
                GO TO SET_TYPE_WALK_UP;
                END;
            ELSE DO;
                C_STRING=C_STRING || '3,0';
                TYPE_EXP=ROW_VEC;
                GO TO SET_TYPE_WALK_UP;     END;
            END;
        /* ROW VECTOR*COLUMN VECTOR */
        IF RNODE->TYPE_EXP=COL_VEC THEN DO;
            IF RESULT_TYPE=COL_VEC THEN DO;
                A_STRING=A_STRING || '3,0';
                B_STRING=B_STRING || '2,0';
                C_STRING=C_STRING || '3,#ROW' || DIGIT_STRINGS(
                 CURRENT_ROW#);
                TYPE_EXP=COL_VEC;
                GO TO SET_TYPE_WALK_UP;
                END;
            END;
```

```
        ELSE GO TO CODER_ILLEGAL;
    /* ROW VECTOR*SCALAR */
    IF RNODE->TYPE_EXP=SCALAR THEN DO;
        A_STRING=A_STRING || '3,0';
        B_STRING=B_STRING || '0,0';
        IF RESULT_TYPE=MATRIX THEN
            C_STRING=C_STRING || '3,#ROW' || DIGIT_STRINGS(
             CURRENT_ROW#);
        ELSE C_STRING=C_STRING || '3,0';
        GO TO SET_TYPE_WALK_UP;   END;
    END;

IF LNODE->TYPE_EXP=COL_VEC THEN DO;
    /* COLUMN VECTOR*ROW VECTOR */
    IF RNODE->TYPE_EXP=ROW_VEC THEN DO;
        IF RESULT_TYPE=COL_VEC THEN GO TO COL_RESULT;
        ELSE IF RESULT_TYPE=ROW_VEC THEN GO TO ROW_RESULT;
        ELSE IF RESULT_TYPE=MATRIX THEN GO TO
            MATRIX_RESULT;
        END;
    /* COLUMN VECTOR*SCALAR */
    IF RNODE->TYPE_EXP=SCALAR THEN DO;
        A_STRING=A_STRING || '2,0';
        B_STRING=B_STRING || '0,0';
        IF RESULT_TYPE=MATRIX THEN
            C_STRING=C_STRING || '2,#COL' || DIGIT_STRINGS(
             CURRENT_COL#);
        ELSE C_STRING=C_STRING || '2,0';
            GO TO SET_TYPE_WALK_UP;    END;
    END;

IF LNODE->TYPE_EXP=SCALAR THEN DO;
    A_STRING=A_STRING || '0,0';
    /* SCALAR*MATRIX */
    IF RNODE->TYPE_EXP=MATRIX THEN DO;
        IF RESULT_TYPE=MATRIX THEN DO;
            B_STRING=B_STRING || '4,0';
            C_STRING=C_STRING || '4,0';
            GO TO SET_TYPE_WALK_UP;    END;
        IF RESULT_TYPE=ROW_VEC THEN DO;
            B_STRING=B_STRING || '3,#ROW' || DIGIT_STRINGS(
             CURRENT_ROW#);
            C_STRING=C_STRING || '3,0';
            TYPE_EXP=ROW_VEC;
            GO TO SET_TYPE_WALK_UP;    END;
        IF RESULT_TYPE=COL_VEC THEN DO;
            B_STRING=B_STRING || '2,#COL' || DIGIT_STRINGS(
             CURRENT_COL#);
            C_STRING=C_STRING || '2,0';
            TYPE_EXP=COL_VEC;
            GO TO SET_TYPE_WALK_UP;    END;
    END;
    /* SCALAR*COLUMN VECTOR */
    IF RNODE->TYPE_EXP=COL_VEC THEN DO;
        B_STRING=B_STRING || '2,0';
        IF RESULT_TYPE=MATRIX THEN
            C_STRING=C_STRING || '2,#COL' || DIGIT_STRINGS(
```

```
                    CURRENT_COL#);
                ELSE C_STRING=C_STRING || '2,0';
                GO TO SET_TYPE_WALK_UP;   END;
            /* SCALAR*ROW VECTOR */
            IF RNODE->TYPE_EXP=ROW_VEC THEN DO;
                B_STRING=B_STRING || '3,0';
                IF RESULT_TYPE=MATRIX THEN
                    C_STRING=C_STRING || '3,#ROW' || DIGIT_STRINGS(
                     CURRENT_ROW#);
                ELSE C_STRING=C_STRING || '3,0';
                GO TO SET_TYPE_WALK_UP;
                END;
        END;

SET_TYPE_WALK_UP:
        OUTPUT_BUFFER=OUTPUT_BUFFER || A_STRING || ',' ||
         B_STRING || ',' || C_STRING || ');'  ;
        CALL SKIP_ANDCOUTPUT;
        CURRENT_NODE->STRING_POINTER=PTR_TO_RESULT_NODE;
        GO TO WALK_UP;

CODER_FUNC_VAR_VAR:
        XPTR1=LNODE->STRING_POINTER;
        XPTR2=RNODE->STRING_POINTER;
        A_STRING=XPTR1->STRINGS;
        B_STRING=XPTR2->STRINGS;
        STRING_POINTER=POINTER_TOCSTRING(A_STRING || B_STRING ||
         ')' , EXPRESSION_AREA);
        TYPE_EXP=SCALAR;
        GO TO WALK_UP;
CODER_FUNC_VAR_OP:
        LOOP_DEPTH=LOOP_DEPTH+1;
        IF RNODE->$TYPE_CODE=-13 THEN GO TO WALK_RIGHT;
        ELSE GO TO CODER_SEP_VAR_OP;

CODER_SEP_OP_OP:
CODER_SEP_OP_VAR:
        IF LNODE->$TYPE_CODE=-13 THEN GO TO WALK_LEFT;
        IF LNODE->TYPE_EXP=MATRIX THEN GO TO MATRIX_WALK_LEFT;
        ELSE IF LNODE->TYPE_EXP=COL_VEC THEN GO TO COL_WALK_LEFT;
        ELSE IF LNODE->TYPE_EXP=ROW_VEC THEN GO TO ROW_WALK_LEFT;
        ELSE GO TO CODER_ILLEGAL;
CODER_SEP_VAR_OP:
        IF RNODE->TYPE_EXP=MATRIX THEN DO;
            ALLOCATE RESULT_STACK;
            RESULT_SEQ=NULL;
            #TEMP_TO_FREE=#TEMP_TO_FREE+1;
            PTR_TO_RESULT_NODE=POINTER_TOCSTRING( '$TEMP2' ||
             DIGIT_STRINGS(MATRIX_TEMP_VARIABLE_#),
             EXPRESSION_AREA);
            MATRIX_TEMP_VARIABLE_#=MATRIX_TEMP_VARIABLE_#+1;
            RESULT_TYPE=MATRIX;
            GO TO WALK_RIGHT;
            END;
        ELSE IF RNODE->TYPE_EXP=COL_VEC THEN GO TO COL_WALK_RIGHT;
        ELSE IF RNODE->TYPE_EXP=ROW_VEC THEN GO TO ROW_WALK_RIGHT;
        ELSE GO TO CODER_ILLEGAL;
```

```
CODER_SEP_VAR_VAR:
      XPTR1=LNODE->STRING_POINTER;
      XPTR2=RNODE->STRING_POINTER;
      A_STRING=XPTR1->STRINGS;
      B_STRING=XPTR2->STRINGS;
      STRING_POINTER=POINTER_TOCSTRING(A_STRING || ',' ||
       B_STRING , EXPRESSION_AREA);
      GO TO WALK_UP;

CODER_EQUL_VAR_OP:
      IF RNODE->$TYPE_CODE=-10 THEN GO TO WALK_RIGHT;
      ALLOCATE RESULT_STACK;
      #TEMP_TO_FREE=#TEMP_TO_FREE+1;
      PTR_TO_RESULT_NODE=LNODE->STRING_POINTER;
      RESULT_TYPE=LNODE->TYPE_EXP;
      RESULT_SEQ=LNODE->SEQ_#_PTR;
      GO TO WALK_RIGHT;
CODER_EQUL_VAR_VAR:
      XPTR1=LNODE->STRING_POINTER;
      XPTR2=RNODE->STRING_POINTER;
      A_STRING=XPTR1->STRINGS;
      B_STRING=XPTR2->STRINGS;
      IF LNODE->TYPE_EXP=SCALAR & RNODE->TYPE_EXP=SCALAR THEN DO;
          CHAR1=A_STRING;   CHAR2=A_STRING;
            /* IF IT IS AN OL/2 SCALAR */
          IF CHAR1='$' & CHAR2¬='$T' THEN GO TO OL2ASGN;
          OUTPUT_BUFFER=OUTPUT_BUFFER ||
              A_STRING || '=' || B_STRING || ';'    ;
          CALL SKIP_ANDCOUTPUT;
          GO TO WALK_UP;
          END;
      IF ASSIGNED(LEVEL#)=0 THEN DO;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       FOR SIMPLE ASSIGNMENT OR UNFINISHED MULTIPLE ASSIGNMENT
       STATEMENT THEN OUTPUT CODE
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
OL2ASGN:
          IF LNODE->SEQ_#_PTR=NULL THEN LSEQ='0';
          ELSE DO;
              SP=LNODE->SEQ_#_PTR;
              LSEQ=STRINGS;   END;
          IF RNODE->SEQ_#_PTR=NULL THEN RSEQ='0';
          ELSE DO;
              SP=RNODE->SEQ_#_PTR;
              RSEQ=STRINGS;   END;
          OUTPUT_BUFFER='CALL @OLASGN(' || A_STRING || ','
              || LSEQ || ',0,1,0'
              || B_STRING || ',' || RSEQ || ',' || RTRANSPOSE(
              LEVEL#) || ',1,' || RNEGATE(LEVEL#) || ');'  ;
          CALL SKIP_ANDCOUTPUT;          END;
      STRING_POINTER=LNODE->STRING_POINTER;
      GO TO WALK_UP;

WALK_LEFT: ALLOCATE TRAVERSE_STK;
      TRAVERSE_STK=CURRENT_NODE;
      CURRENT_NODE=CURRENT_NODE->LLINK;
      LEVEL#=LEVEL#+1;
```

```
        GO TO NEXT_NODE;
WALK_RIGHT: ALLOCATE TRAVERSE_STK;
        TRAVERSE_STK=CURRENT_NODE;
        CURRENT_NODE=CURRENT_NODE->RLINK;
        LEVEL#=LEVEL#+1;
        GO TO NEXT_NODE;

 WALK_UP:
        IF ¬ALLOCATION(TRAVERSE_STK) THEN GO TO LEAVE_CODER;
        CURRENT_NODE->$TYPE_CODE=0;
        LTRANSPOSE(LEVEL#),RTRANSPOSE(LEVEL#)='0';
        LNEGATE(LEVEL#),RNEGATE(LEVEL#)='0';
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         OUTPUT PARTITIONING LOOP END STATEMENTS
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        DO WHILE (#CEND_STATEMENTS>0);
            OUTPUT_BUFFER='END;';
            CALL SKIP_ANDCOUTPUT;
            #CEND_STATEMENTS=#CEND_STATEMENTS-1;
            CURRENT_COL#=CURRENT_COL#-1;
            LOOP_DEPTH=LOOP_DEPTH-1;    END;
        DO WHILE (#REND_STATEMENTS>0);
            OUTPUT_BUFFER='END;';
            CALL SKIP_ANDCOUTPUT;
            #REND_STATEMENTS=#REND_STATEMENTS-1;
            CURRENT_ROW#=CURRENT_ROW#-1;
            LOOP_DEPTH=LOOP_DEPTH-1;    END;
        CURRENT_NODE=TRAVERSE_STK;
        FREE TRAVERSE_STK;
        LEVEL#=LEVEL#-1;
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         STACK TEMPORARY VARIABLES TO BE FREED
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        DO WHILE (#TEMP_TO_FREE>0);
            #FREE(LEVEL#-1)=#FREE(LEVEL#-1)+1;
            FREE_PTR(LEVEL#-1,#FREE(LEVEL#-1))=PTR_TO_RESULT_NODE;
            FREE RESULT_STACK;
            IF ALLOCATION(RESULT_STACK) THEN
            IF FREE_PTR(LEVEL#-1,#FREE(LEVEL#-1))=
             PTR_TO_RESULT_NODE THEN #FREE(LEVEL#-1)=
             #FREE(LEVEL#-1)-1;
            #TEMP_TO_FREE=#TEMP_TO_FREE-1;
          END;
        IF LEVEL#=0 THEN GO TO NEXT_NODE;
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         FREE TEMPORARY VARIABLES WHEN OUTSIDE A SYSTEMS
         PARTITIONING LOOP
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        IF LOOP_DEPTH=0 THEN  DO;
            DO I=LEVEL# TO 15;
                DO WHILE (#FREE(I)>0);
                    XPTR1=FREE_PTR(I,#FREE(I));
                    C_STRING=XPTR1->STRINGS;
                    T_STRING=C_STRING;
                    IF T_STRING='$TEMP' THEN DO;
                        OUTPUT_BUFFER='CALL @OLFSTR(' ||
                            C_STRING || ');'  ;
```

```
                    CALL SKIP_ANDCOUTPUT;
                    END;
                #FREE(I)=#FREE(I)-1;
                END;
            END;
        END;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       OUTPUT CODE COMPILED UP UNTIL NOW IF NOT INSIDE SYSTEMS
       PARTITIONING LOOP
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF LOOP_DEPTH=0 & #LINES¬=0 THEN DO;
        DO I= 1 TO #LINES;
          OUTPUT_BUFFER=OUTLINES(I);
         J=LENGTH(OUTPUT_BUFFER);
        PUT FILE(SYSPUNCH) EDIT (SUBSTR(OUTPUT_BUFFER,
         1,J)) (X(1),A(79));
          PUT FILE(SYSPRINT) LIST (OUTLINES(I)) SKIP; END;
        #LINES=0;  END;
      GO TO NEXT_NODE;

LEAVE_CODER: STK_PTR=STK_PTR-1;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       INSURE THAT ALL STACKS ARE EMPTY
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      DO WHILE (ALLOCATION(RESULT_STACK));
        FREE RESULT_STACK;   END;
      DO WHILE (ALLOCATION(TRAVERSE_STK));
        FREE TRAVERSE_STK;   END;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       FREE ANY REMAINING TEMPORARY VARIABLES
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      DO I= 0 TO 15;
        DO WHILE(#FREE(I)>0);
          XPTR1=FREE_PTR(I,#FREE(I));
          C_STRING=XPTR1->STRINGS;
          T_STRING=C_STRING;
          IF T_STRING='$TEMP' THEN DO;
            OUTPUT_BUFFER='CALL @OLFSTR(' || C_STRING
              || ');';
            CALL SKIP_ANDCOUTPUT;
            END;
          #FREE(I)=#FREE(I)-1;
          END;
        END;
      /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
       OUTPUT ANY REMAINING CODE
       * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
      IF #LINES¬=0 THEN DO;
        DO I= 1 TO #LINES;
          OUTPUT_BUFFER=OUTLINES(I);
         J=LENGTH(OUTPUT_BUFFER);
        PUT FILE(SYSPUNCH) EDIT (SUBSTR(OUTPUT_BUFFER,
         1,J)) (X(1),A(79));
          PUT FILE(SYSPRINT) LIST (OUTLINES(I)) SKIP; END;
        END;
      RETURN;
```

```
    CODER_ILLEGAL:
    MULTOO(4,3): MULTOO(3,3): MULTOO(2,4): MULTOO(2,2):
    MULTOV(4,3): MULTOV(3,3): MULTOV(2,4): MULTOV(2,2):
    MULTVO(4,3): MULTVO(3,3): MULTVO(2,4): MULTVO(2,2):
          PUT LIST ('CODER ILLEGAL') SKIP;
          PUT LIST ('TYPE_CODE=' , $TYPE_CODE, 'TYPE_EXP=',
                TYPE_EXP) SKIP;
          PUT DATA (TYPE_RIGHT, TYPE_LEFT)  SKIP;
          GO TO LEAVE_CODER;

          /* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
           SAVE GENERATED CODE
           * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    SKIP_ANDCOUTPUT:  PROCEDURE;
          #LINES=#LINES+1;
          OUTLINES(#LINES)=OUTPUT_BUFFER;
          OUTPUT_BUFFER = ''              ;
        END SKIP_ANDCOUTPUT;

    POINTER_TOCSTRING: PROCEDURE(STRING,AREA) RETURNS(POINTER);
          DECLARE STRING CHARACTER (200) VARYING , AREA AREA (*)   ;
          STRING_LENGTH = LENGTH ( STRING )          ;
          ALLOCATE VARIABLE_STRING IN ( AREA )  ;
          STRINGS = STRING         ;
          RETURN ( SP )              ;
        END POINTER_TOCSTRING;

          /* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
           SET UP OPERAND NAMES AND SEQUENCE #, TRANSPOSE, &
           NEGATE INDICATORS
           * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    SET_VARIABLES: PROCEDURE;
          XPTR1=LNODE->STRING_POINTER;
          XPTR2=RNODE->STRING_POINTER;
          A_STRING=XPTR1->STRINGS;
          B_STRING=XPTR2->STRINGS;
          C_STRING=PTR_TO_RESULT_NODE->STRINGS;
          IF LNODE->SEQ_#_PTR=NULL THEN LSEQ='0';
          ELSE DO;
              SP=LNODE->SEQ_#_PTR;
              LSEQ= STRINGS;   END;
          IF RNODE->SEQ_#_PTR=NULL THEN RSEQ='0';
          ELSE DO;
              SP=RNODE->SEQ_#_PTR;
              RSEQ=STRINGS;   END;
          IF RESULT_SEQ=NULL THEN RSLTSEQ='0';
          ELSE DO;
              SP=RESULT_SEQ;
              RSLTSEQ=STRINGS;   END;
          A_STRING=A_STRING || ',' || LSEQ || ',' ||
              LTRANSPOSE(LEVEL#) || ',1,' || LNEGATE(LEVEL#) || ',';
          B_STRING=B_STRING || ',' || RSEQ || ',' ||
           RTRANSPOSE(LEVEL#) || ',1,' || RNEGATE(LEVEL#) || ',' ;
          C_STRING=C_STRING || ',' || RSLTSEQ || ','  ;
          ASSIGNED(LEVEL#-1)=1;
        END SET_VARIABLES;
END CODER;
```

APPENDIX D

OL/2 ARRAY EXPRESSION EXAMPLES

The first five examples are those used in the text. The first is used in sections 3 and 4, and the others occur in section 4. Example 6 shows that if parenthesis are not used in example 5, less temporary storage would result. The seventh statement shows that the precedence of the multiply operation is context dependent. Notice that the expression, because of the precedence relations, is parsed as R = X*((Y'*A)*B) and this requires the least number of operations. Statements 8 and 9 illustrate how a sequence of multiplications or additions is compiled for array operands.

EXAMPLE 1: R=A*B+C*(D+E);

```
DO #COL1=$1R1->#LOWER(2) TO $1R1->#UPPER(2);
CALL @OLMULT($1A1,0,0,1,0,4,0,$1B1,0,0,1,0,2,#COL1,$TEMP10,0,2,0);
CALL @OLADD($1D1,0,0,1,0,2,#COL1,$1E1,0,0,1,0,2,#COL1,$TEMP12,0,2,0);
CALL @OLMULT($1C1,0,0,1,0,4,0,$TEMP12,0,0,1,0,2,0,$TEMP11,0,2,0);
CALL @OLADD($TEMP10,0,0,1,0,2,0,$TEMP11,0,0,1,0,2,0,$1R1,0,2,#COL1);
END;
CALL @OLFSTR($TEMP11);
CALL @OLFSTR($TEMP10);
CALL @OLFSTR($TEMP12);
```

EXAMPLE 2: R=A+B;

```
CALL @OLADD($1A1,0,0,1,0,4,0,$1B1,0,0,1,0,4,0,$1R1,0,4,0);
```

EXAMPLE 3: R=ALPHA+BETA*GAMMA*B;

```
CALL @OLMULT(ALPHA+BETA*GAMMA,0,0,1,0,0,0,$1B1,0,0,1,0,4,0,$1R1,0,4,0);
```

EXAMPLE 4: R=ALPHA+BETA*GAMMA*(X,Y)*B;

```
#TEMP00=@OLIPRD($1X1,0,0,1,0,$1Y1,0,0,1,0);
CALL @OLMULT(ALPHA+BETA*GAMMA*#TEMP00,0,0,1,0,0,0,$1B1,0,0,1,0,4,0,$1R1,0,4,0);
```

EXAMPLE 5: R=ALPHA*((A*B)*(C*D));

```
CALL @OLMULT($1A1,0,0,1,0,4,0,$1B1,0,0,1,0,4,0,$TEMP20,0,4,0);
DO #COL1=$1R1->#LOWER(2) TO $1R1->#UPPER(2);
CALL @OLMULT($1C1,0,0,1,0,4,0,$1D1,0,0,1,0,2,#COL1,$TEMP11,0,2,0);
CALL @OLMULT($TEMP20,0,0,1,0,4,0,$TEMP11,0,0,1,0,2,0,$TEMP10,0,2,0);
CALL @OLMULT(ALPHA,0,0,1,0,0,0,$TEMP10,0,0,1,0,2,0,$1R1,0,2,#COL1);
END;
CALL @OLFSTR($TEMP10);
CALL @OLFSTR($TEMP11);
CALL @OLFSTR($TEMP20);
```

EXAMPLE 6: R=ALPHA*A*B*C*D;

```
DO #COL1=$1R1->#LOWER(2) TO $1R1->#UPPER(2);
CALL @OLMULT($1C1,0,0,1,0,4,0,$1D1,0,0,1,0,2,#COL1,$TEMP12,0,2,0);
CALL @OLMULT($1B1,0,0,1,0,4,0,$TEMP12,0,0,1,0,2,0,$TEMP11,0,2,0);
CALL @OLMULT($1A1,0,0,1,0,4,0,$TEMP11,0,0,1,0,2,0,$TEMP10,0,2,0);
CALL @OLMULT(ALPHA,0,0,1,0,0,0,$TEMP10,0,0,1,0,2,0,$1R1,0,2,#COL1);
END;
CALL @OLFSTR($TEMP10);
```

```
 CALL @OLFSTR($TEMP11);
 CALL @OLFSTR($TEMP12);
```

EXAMPLE 7: R=X*Y'*A*B;

```
 CALL @OLMULT($1Y1,0,1,1,0,3,0,$1A1,0,0,1,0,4,0,$TEMP11,0,3,0);
 CALL @OLMULT($TEMP11,0,0,1,0,3,0,$1B1,0,0,1,0,4,0,$TEMP10,0,3,0);
 CALL @OLMULT($1X1,0,0,1,0,4,0,$TEMP10,0,0,1,0,4,0,$1R1,0,4,0);
 CALL @OLFSTR($TEMP11);
 CALL @OLFSTR($TEMP10);
```

EXAMPLE 8: R=A*B*C*D;

```
 DO #COL1=$1R1->#LOWER(2) TO $1R1->#UPPER(2);
 CALL @OLMULT($1C1,0,0,1,0,4,0,$1D1,0,0,1,0,2,#COL1,$TEMP11,0,2,0);
 CALL @OLMULT($1B1,0,0,1,0,4,0,$TEMP11,0,0,1,0,2,0,$TEMP10,0,2,0);
 CALL @OLMULT($1A1,0,0,1,0,4,0,$TEMP10,0,0,1,0,2,0,$1R1,0,2,#COL1);
 END;
 CALL @OLFSTR($TEMP10);
 CALL @OLFSTR($TEMP11);
```

EXAMPLE 9: R=A+B+C+D;

```
 CALL @OLADD($1A1,0,0,1,0,4,0,$1B1,0,0,1,0,4,0,$1R1,0,4,0);
 CALL @OLADD($1R1,0,0,1,0,4,0,$1C1,0,0,1,0,4,0,$1R1,0,4,0);
 CALL @OLADD($1R1,0,0,1,0,4,0,$1D1,0,0,1,0,4,0,$1R1,0,4,0);
```